Examensarbete

Dynamically Adaptive Intelligent Agents in Driving Simulator Environments

av

Linus Gustavsson

LITH-IDA-EX—07/059—SE

2007-11-16

Linköpings Universitet
Institutionen för datavetenskap

Examensarbete

Dynamically Adaptive Intelligent Agents in Driving Simulator Environments

av

Linus Gustavsson

LITH-IDA-EX—07/059—SE

2007-11-16

Handledare och Examinator: Rita Kovordanyi

Linköpings Universitet
Institutionen för datavetenskap

Avdelning, institution
Division, department

Institutionen för datavetenskap

Department of Computer
and Information Science

Datum
Date

2007-11-16

Linköpings universitet

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport

_____

ISBN —

ISRN LITH-IDA-EX—07/059—SE

Serietitel och serienummer
Title of series, numbering

ISSN —

URL för elektronisk version

www.ep.liu.se

Titel
Title

Dynamically Adaptive Intelligent Agents in Driving Simulator Environments

Författare
Author

Linus Gustavsson

Sammanfattning
Abstract

In this thesis work I have been working with two traffic simulators called Hank and ST Software. Hank is a research tool at the University of Iowa and ST Software is a commercial product. To evaluate which of these is the most suitable for behavior research I have implemented three types of intelligent agents: Overtaking Agent, Traffic Light Agent and Meeting Agent. The thesis work was extended by adding the possibility for realistic human behavior to the agents.

The result indicated that Hank allowed for greater control over behavior while ST Software allowed for faster and easier implementation.

Nyckelord
Keywords

Traffic Simulator, Agent, Hank, ST Software, Overtaking, Traffic Light, Meeting, Behavior Research, EDF, Human Behavior

Abstract

In this thesis work I have been working with two traffic simulators called Hank and ST Software. Hank is a research tool at the University of Iowa and ST Software is a commercial product. To evaluate which of these is the most suitable for behavior research I have implemented three types of intelligent agents: Overtaking Agent, Traffic Light Agent and Meeting Agent. The thesis work was extended by adding the possibility for realistic human behavior to the agents.

The result indicated that Hank allowed for greater control over behavior while ST Software allowed for faster and easier implementation.

Sammanfattning

I det här examensarbetet har jag arbetat med två olika trafiksimulatorer vid namn Hank och ST Software. Hank är ett forskningsverktyg på Iowas universitet och ST Software är en kommersiell produkt. För att utvärdera vilken av dessa två är mest lämpad för beteende-forskning så har jag implementerat tre typer av intelligenta agenter: Omkörningsagent, Trafikljusagent och en Mötesagent. Examensarbetet utökades genom att lägga till möjligheten för realistiskt, mänskligt beteende till agenterna.

Resultatet visade på att Hank gav mer kontroll över beteende medan ST Software tillät snabbare och enklare implementeringar.

Contents

1 Introduction 1
1.1 Background 1
1.2 The goal of this study 1
1.3 Future goals 2
2 Basic Theory and Formulas 3
2.1 Distance Formulas 3
3 Strategic algorithms for the agents 9
3.1 Basic outline for the agents 9
3.1.1 Overtaking 9
3.1.2 Traffic Light 9
3.1.3 Meeting at a predetermined point 10
3.2 Extended outlines for agents 11
3.2.1 Overtaking AI 11
3.2.2 Traffic Light AI 12
3.2.3 Meeting AI 12
3.3 Configuration 13
4 Software choices for designing the agents 15
4.1 Introduction 15
4.2 Logical Representation of the worlds 15
4.2.1 Environment Description Framework (EDF) 16
4.2.2 ST Software network files (NET) 17
4.2.3 OpenDrive 18
4.3 Comparison ST Software vs. Hank 18
4.3.1 Language 20
4.3.2 User's Manual 21
4.3.3 Controls 22
4.3.4 Worlds 22
4.3.5 Hardware 23
4.3.6 Agent Designs 23
4.3.7 Modules 24
4.3.8 Overview 24
4.4 Conclusion 25
5 Implementation of simple agents 27
5.1 Creating a world 27
5.2 Populating the world 29
5.3 Overtaking agent 33
5.4 Traffic Light Agent 35
5.5 Meeting Agent 37
6 Agents structures in ST Software 39
6.1 Overtaking Agent 39
6.2 Traffic Light Agent 39
6.3 Meeting Agent 40
7 Agents structures in Hank 41

7.1 Notes about Hank 41
7.2 Overtaking agent 41
7.3 Traffic Light Agent 42
7.4 Meeting Agent 42
8 Adding a feeling of realism 43
8.1 Traffic light behavior 43
8.2 Passing behavior 44
8.3 Suggested variables 46
9 Results 49
9.1 Results using ST Software 49
9.1.1 Overtaking 49
9.1.2 Overtaking Behavior 56
9.1.3 Traffic Light 57
9.1.4 Traffic Light Behavior 61
9.1.5 Meeting 62
9.2 Results using Hank 67
9.2.1 Overtaking - Background 68
9.2.2 Files that were central in Hank for agents 69
9.2.3 The implementation of an Overtaking agent in detail 70
9.2.4 The end result 77
10 Future Work 79
10.1 Overtaking Agent 79
10.2 Traffic Light Agent 79
10.3 Meeting Agent 80
10.4 Hank 81
11 References 83

List of Figures

Figure 1 Illustration of Jenkin and Rilett's (2006) idea notion of distance during overtaking 6
Figure 2 The different parts of the ST Software Simulator 18
Figure 3 The different parts of the Hank Simulator 19
Figure 4 StControl which allows keeping track of a lot of data during simulation 19
Figure 5 The Hank design tool (Hank source code in C-language) 20
Figure 6 An example on how ST Scenario scripting language only need a text editor to design behavior. 21
Figure 7 An image displaying the ST RoadDesign tool for creating worlds 23
Figure 8 The main screen during simulation in ST Software's simulator 25
Figure 9 The main screen of Hank during simulation (DOS window is invisible when not debugging) 26
Figure 10 Zoomed in image of an intersection in the ST RoadDesign tool. 28
Figure 11 A photo of one of the four stations of ST Software's simulator at Linköping University. 49
Figure 12 An overview of the scheme for the ST Software based overtaking agent 56
Figure 13 An overview of the scheme for the traffic light agent in ST Software 60
Figure 14 Three smaller schemes describing the meeting agent in ST Software as well as how it is selected and sometimes adjusted 66
Figure 15 The much simpler workstation for experiments with Hank. 67

List of Tables

Table 1 Overtaking Speeds and Distances 33
Table 2 Average test results for needed distance during overtaking 34
Table 3 Values from using advanced equations for overtaking 34
Table 4 Typical Swedish Traffic Light Durations 36
Table 5 Test results on difference at arrival time for meeting agent 66

1 Introduction

The purpose of this thesis is to decide which simulator the Department of Computer and Information Science at Linköping University should focus on in the future for research on agent designs. You could call it an important first step towards designing realistic agents within traffic simulators at the university.

1.1 Background

The reason we chose ST Software as one of the simulators to test was because the university recently purchased a set of four stations. Originally we had another simulator than Hank in mind, but because of some problems with making a deal to work with it we decided to go with Hank instead. The main reason was that both the third simulator and Hank were running EDF (Environment Description Framework) as their logical representation system. The University of Iowa agreed to let us work with Hank.

1.2 The goal of this study

Since the purpose was to figure out which simulator to use in the future we decided on two goals. The first goal of this study was to evaluate which out of ST Software and Hank would be the best option for implementing and testing various types of intelligent agents for emulating realistic traffic scenarios. This would be done by implementing three agents. The types were one agent for performing a passing maneuver, one for interacting with traffic lights and one for meeting up at a desired point with another human driver. By designing these three agents I expected to run into different limits in the simulators which would help me judge which simulator that is the optimal choice for working with agent designs.

A secondary goal was to adjust these three agent types to have a more human behavior apart from simply being able to perform their tasks. A normal human has many attributes that could and preferably should be implemented in an intelligent agent. Some suggestions for names on these variables could be Aggressive, Inattentive or Stressed. An aggressive driver would take more risks while an unobservant driver could take risks without knowing they do. From personal experience stressed would involve both since many humans drive more aggressively when stressed and might also miss things because their minds are somewhere else.

During the work it became clear that there is more to designing agents than “can it or can it not be done”. Because of this the evaluation changed a bit from focusing on which simulator could design the best agent behavior to a broader evaluation where I observed many different things for example: overview of code, controls, world designing, support, user information and other details that affected the suitability of the two simulating environments.

1.3 Future goals

In the future the Department of Computer and Information Science at Linköping University hope to expand these three agents into more advanced and realistic ones that will interact with human drivers in a simulated environment, and can be used for training self learning agents and to test human behavior in realistic traffic situations.

2 Basic Theory and Formulas

2.1 Distance Formulas

A starting point on developing intelligent agents for driving simulation is specification of distance formulas. Of central interest are those that capture time to travel a certain distance, or the distance we have to travel to arrive at a meeting point.

A time-to-collision formula for cars that the own car catches up with (this formula could also be used if the car ahead is slowing down) (Shladover & Tan, 2006)

$$T = \frac{(y - x)}{(\dot{y} - \dot{x})} \qquad \textbf{Equation 2.1}$$

The variable *y* stands for the driver's car's position along the road and *x* the position of the car in front of the driver—both measured in meters. Below In the denominator part is the corresponding derivate, in other words speed of the cars. The result *T* denotes time to collision (TTC) measured in seconds.
One thing to observe here is that one could encounter problems with division by zero if both cars have the same speed. This means that it is a slightly dangerous formula in where that it can quickly go from one extreme state into another.
Note also that this formula should work also for meeting cars, in other words, if one of the speeds is negative. Meeting does not necessarily mean collision—the cars could be in different lanes when they reach the collision time.

An intersection meeting formula to avoid collisions from left turnings is as follows (Shladover & Tan, 2006)

$$T = \frac{y}{\dot{y}} - \frac{x}{\dot{x}} \qquad \textbf{Equation 2.2}$$

The variables are the same as in Equation 2.1, including the result *T* in seconds. This equation is for calculating the time difference between two objects who are heading for a specific point. In other words, it gives the time for how long one would have to wait for the other car if they both planned on stopping at the same place. This might be good if one wants to know the time difference between two arrivals, but it will not give the time of arrival. This equation also can not be used for static objects because of the division with zero, but on the other hand there is a set point they will both pass, and if one is not moving at all, then it will never reach that point anyway. One could treat it as a special case in that distance x is 0 and its speed is 0, which would give a zero divided with zero equation, which one could set to zero as a special case to avoid problems. Then maybe we would have a time for arrival.

Safe following distance could be described using the following formula (Abe & Richardson, 2006)

$$D_w = V_f * Rt + \frac{V_f^2}{2D_f} - \frac{V_l^2}{2D_l} \quad \textbf{Equation 2.3}$$

This formula calculates safe distance *D* to the car including variables like *V* for speed *Rt* for reaction time and *D* as deceleration. The index *f* is for following car, and *l* is leading car. We would want to keep the distance to feel like we drive at a distance that we can easily break in time for. Probably be useful for making a good agent to never collide with the test driver.

$$D_s = \frac{V_f^2}{2D_f} + \tau V_f + \delta \quad \textbf{Equation 2.4}$$

Another way to calculate safe following distance is given in Equation 2.4 (Cheng & Fujioka 1998). Again *f* is for the following car, and *tau* here stands for basically the reaction time. *Delta* stands for safety margin, or in other words the remaining distance between the two cars when they both have stopped. This equation is similar to Equation 2.3, except it does not use the information of the leading car.

$$D_4 = D_s + \frac{(V_b - V_{own})^2}{2a} + \delta \quad \textbf{Equation 2.5}$$

If we use Equation 2.4 and modify it slightly we have an equation for safe passing of a car in a multi lane scenario in which we have a car behind us in the lane we want to use for overtaking the car ahead of us. In other words the safe distance for performing this task is given by the variables *D* from earlier, *V* which is the speeds, *a*, which is the acceleration required for passing the car ahead of us and *delta* is the safety margin. The *b* stands for car behind agent and *own* stand for the agent itself.
So what we have here is basically an overtaking equation but with another car coming from behind instead of from ahead of us. The question is if we can use this equation with maybe a slight change for meeting cars. Instead of subtracting velocities we would end up adding the speeds since we would be meeting them instead of driving away from them.

$$D_1 = \frac{V_b^2 - V_{own}^2}{2D_{own}} + \tau V_b + \delta \quad \textbf{Equation 2.6}$$

Equation 2.6 is an auxiliary to the previous one. It determines safe distance to the car behind us in case we have to brake during the overtaking. All variables as described in previous equations.

We have a few variables optimized with regards to a good passing maneuver path by based on empirical tests (Shamir 2004)

$$D \approx 2.4V\sqrt{\frac{W}{A}}$$

$$T \approx \sqrt{3}\frac{W^{3/2}\sqrt{A}}{V^2} + 2.4\sqrt{\frac{W}{A}}$$

In which W is the width of the lane, V is the speed of the overtaking agent, A is the max acceleration of the same agent. D is distance and T is the time for driving up alongside with the car in front.

Two other equations needed before the final results are the following two:

$$T_b = \frac{(L + L_1)}{(V - V_1)}$$

$$D_b = V\frac{(L + L_1)}{(V - V_1)}$$

L is the length of the passing car and L indexed *1* is length of the car in front. In the same manner V is the passing cars speed and V indexed *1* is the speed of the car in front. D in this equation is then the absolute distance needed to travel and T is the time needed for an overtake maneuver.

This gives us the minimum time and minimum distance of

$$T_{\min} = 2T + T_b \qquad \textbf{Equation 2.7}$$

$$D_{\min} = 2D + D_b \qquad \textbf{Equation 2.8}$$

If these formulas prove to work well enough we have the ability to determine time and distance needed to perform the overtaking.

Jenkins and Rilett (2006) mention several popular equations to keep following a car at a safe distance. A very logical thing that Jenkins and Rilett (2006) mention is that the equations' performance all depend on how the simulator is designed. Some might work well on one system, while another equation might be more reliable on another system. Earlier, we listed some lane changing equations (Equation 2.3 to Equation 2.6) and Jenkins and Rilett (2006) have a few additional ones.

For example there is an equation for when there is a following equation captures forced lane change at places along the road where two lanes turn merge into a single lane.

$$a_{accept} = a_{\min} + (e - a_{\min})\sqrt{\frac{O_E}{O_L}} \qquad \textbf{Equation 2.9}$$

To enter and exit the freeway, the level of risk (assumed to be denoted by *a* index *accept*) is calculated using the minimum acceptable deceleration, a_{min}, the emergency deceleration rate, *e*, the distance to the end of the opportunity, O_E and the length of the opportunity, O_L.

It is uncertain if this formula can be used for anything in the three intelligent agent scenarios we are planning to look closer into. There are a few more similar equations mentioned but we will move on to the more interesting ones involving overtaking, or in other words changing lane into a lane with meeting traffic.

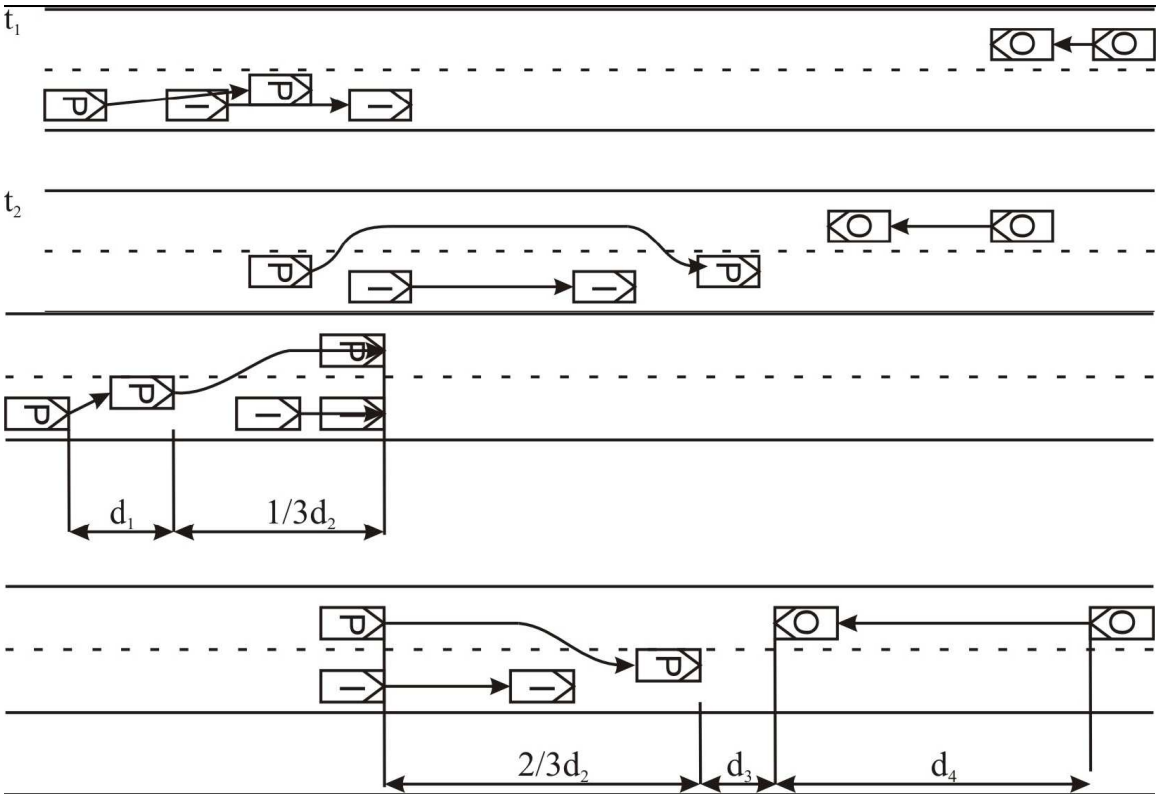

Figure 1 Illustration of Jenkin and Rilett's (2006) idea notion of distance during overtaking

The above pictures are taken from Jenkins and Rilett (2006) and show a classic passing maneuver and illustrate what the later distance equations below mean. Agents with the ability to perform passing maneuvers are not that common in simulators yet. A review by Hoban and McLean (1982) spoke of only five simulators supporting it at that time. These are abbreviated as SOVT, ROADSIM, SOFOT, TRARR and last we have a reference to a Swedish program by the Swedish National Road and Transport Research Institute (VTI). According to the same review SOFOT and TRARR incorporate deterministic rules for passing whereas SOVT, TWOWAF/TWOPAS, and VTI use gap acceptance probabilities derived from extensive field studies. Jenkins and Rilett (2006) did not explain what these abbreviations stood for so you will have to accept them in their short form only. Since then new simulators have appeared like ST Software which is part of this thesis.

A formula for safe distance to begin a passing maneuver is as follows:

$$d_1 = 0.278 t_1 (\bar{s}_p - m_{pi} + \frac{\bar{a}_p t_1}{2}) \quad \textbf{Equation 2.10}$$

t is for time to decide to perform the maneuver, *s* average speed, *a* average acceleration, m is speed difference between passing and impeding vehicle. (expressed in km/h.)

$$d_2 = 0.278 t_2 \bar{s}_p \quad \textbf{Equation 2.11}$$

This part calculates the actual distance needed to pass the impeding vehicle.

Jenkins and Rilett (2006) did not list any more equations, but they mentioned that one of course needs to calculate how far the car traveling in the opposite direction will travel in the same time, and take into account the distance when they started so we can see how much distance there will be between them when the maneuver is over. The distance *d4* is about 2/3 of *d2* assuming both the passing car and the meeting car have the same speed.

These distances are referred to as “Sight Distances” and were used to mark areas of segments of roads as possible passing areas according to Jenkins and Rilett (2006). So in their case it appears like it was pre-calculated zones for safe overtaking with these formulas, which would mean that these formulas might not be suitable to use during runtime.

3 Strategic algorithms for the agents

3.1 Basic outline for the agents

Let us start with listing what is required for each agent to function. Later we will go into details on what other events and surroundings could be desirable for an agent to be able to adapt to.

3.1.1 Overtaking

This behavior has the most amounts of equations which means that we have several options to choose from that can be adequate for the agent. At this early stage we will just focus on the simple basic function to make a car pass another car without driving into any other car. A simple algorithm for this would look like something along the lines of these points.

- Detect the closest car agent driving the opposite direction ahead of me.
- Determine that cars speed
- Detect which car I need to pass ahead of me.
- Determine that cars speed.
- Calculate if it is safe to pass.
    - Calculate how long it takes to pass and how long it takes to meet the other car, and then determine if the difference is above a safety threshold. This could be done with (Equation 2.1) or (Equation 2.7)
    - Alternatively there are formulas that let the agent decide on a safety distance and then check against this distance to decide if the agent can perform an overtaking or not. For example an edited version of (Equation 2.5) to work with meeting cars instead of following cars or maybe (Equation 2.8)

One could compare my ideas with a list made by Henry McLoughlin et al (1993) that in the book Generic Intelligent Driver Support (p.103) lists what they consider important for overtaking. The list is more a listing of different cases, which I have tried to rewrite into different parts to observe.

- Time until the approaching car meets us
- There is a safe gap in front of the car ahead of us
- Can we accelerate enough to overtake even if distance is less than wanted?
- Distance to intersection is long enough to also include safe breaking before it.

3.1.2 Traffic Light

Information about traffic lights was scarce. Most of the available literature treated areas like visual detection of cars and other traffic, while some articles focused on how to make

intersections work smoother by examining the probability that a car comes to an intersection. There was some information regarding the logical representation of traffic lights for the EDF format in Willemsen (2000). It has a few states, red, yellow, green flashing yellow. It sounds like there is a separate traffic light behavior that sets these values. EDF seems to already feature a rather accurate version of a dynamic and adjustable traffic light. It is designed to be able to adjust its cycle to display the desired color when a road-user reaches the traffic light. The method that EDF's traffic light system uses is that it calculates how long it expects an agent to take to reach the traffic light, and then informs the traffic light what part of the cycle we desire when the agent reaches it. This is done repeatedly while approaching to compensate for speed changes. This is helpful for using EDF, but it is uncertain how it would help in a different logic representation. Something that does help for different logic representation is that the same thesis about EDF (Willemsen 2000) also has a section about traffic lights a bit more in general. The traffic light has a normalized cycle time and a target time. Adjustments to the cycle time are done by uniformly expanding or compressing it. The code for the sequencer is available in SDL (Scenario Description Language) code.

From the information available I believe a simple algorithm might be able to use one of the earlier equations. The one that seemed most fitting and simple was Equation 2.1 where one agent would have zero speed. The steps for the traffic light would be something similar to:

- Detect time until car reaches traffic light
- Adjust traffic light cycle according to plans and time left

A different approach would be to have some form of agent detectors that detect when the driver comes close enough to the traffic light and adjust it after that. This could become troublesome since it might not continuously check the cars speed and the cycle time might not be adjusted well enough.

3.1.3 Meeting at a predetermined point

The principle of having two agents meet is not very different from having a static traffic light calculate when a car reaches its position. We should be able to use the same formula for both most likely. Since we start with focusing on the simplest event we are going to assume just two cars that are going to meet.

3.2 Extended outlines for agents

Here follows some ideas on special cases that could be interesting to be able to handle to make the agents perform more human-like.

3.2.1 Overtaking AI

In reality we would like a much larger and more complex agent system that handles all common and some uncommon traffic situations. The overtaking AI for example should be able to solve several problems.

- Is any car behind me showing that they are going to overtake?
- Are there several lanes and do I need to observe cars behind me in other lanes?
- How far can I see? Can I see the closest car in opposite direction?
- Is the car in front of me far to the left or right or maybe in the middle?
- What are all cars accelerations? And how do the accelerations change?
- Unexpected obstacles on the road that might make people change lateral position on the road.
    - Aborting the overtaking
    - Alternatively never start the overtaking in unsafe zones.
- Any special traffic symbols close by that may affect the agent?
    - Intersections were people might turn left, or turn out into the road the agent is traveling.
    - Change in speed limit
    - Road lines
    - Poor sight? (like at a zebra crossing or bus stop)

Some notes from Jenkins and Rilett (2006) regarding how they thought about the visual field of the agent planning to do a passing maneuver.

- The impeding vehicle travels at constant speed
- Time to determine whether the opposing lane is clear and begin the maneuver is important to know.
- Passing vehicle accelerates at the start and then continue on constant speed
- The speed difference between passing and impeding vehicle during the maneuver is 15 km/h

They chose to not take into account if the vehicle they are passing will change speed. This is correct according to how people should drive, but in reality it often happens that people getting overtaken start to notice their own speed and speed up a bit.

3.2.2 Traffic Light AI

Traffic lights can get relatively advanced if one tries to cover everything.

- What kind of obstacles are on the way to the traffic light?
  - Speed signs, Stop signs, other traffic lights.
  - Cars
- Visual field of driver.
  - Is it a straight road so the incoming driver will see if the traffic light acts funny? In such a case maybe we will have to start tinkering with the light before it comes into sight.
- Want to keep the cycle look natural while approaching
  - How do we adjust the cycle time to still have the right color when driver reaches it?

3.2.3 Meeting AI

Just as stated in the simple version, a system to make agents meet at a specific intersection or location should be very similar to the traffic light system. However it increases in complexity when the amount of cars supposed to meet up increase.

- Try to coordinate all moving cars to reach the point at the same time
- Obstacles on the way? (see Traffic Light)
- How far can a driver see? (Can we place a car out of sight?)
- Removing cars that are not supposed to be around at the designated time for the meeting.
  - Remove cars out of sight?
  - Let cars avoid the place by turning in an earlier intersection?
    - How do we avoid making the road seem too empty?
- Possibly determine which agent that is the best choice for the meeting.
  - Closest? Same distance as Driver? Least obstacles in the way? Create a new car?
- How does the driver's acceleration and change of acceleration affect the meeting?
- The Driver makes a wrong turn
  - What will the new meeting time be?
  - How do we know which way the Driver is going in the first place?
    - Shortest distance?
    - Widest road?
    - Following road signs marking a predetermined route, such as if they should head through central city or take the highway around to the other end of the city.
  - Can we force the driver to not go a path we don't want?
    - Traffic lights or cars in the way of alternate paths?

3.3 Configuration

As a final note on all three agent categories there is also a need for changing behavior of the agents, preferably by sliders. Sometimes we could have a need for an agent that ignores for example red light to achieve a specific event or maybe one that tries to pass the driver even if the distance to the car traveling in the opposite direction is too short. Hopefully we will have a few suggestions for what things can be controlled by sliders and what would be too much work compared to the result in the end.

4 Software choices for designing the agents

4.1 Introduction

ST Software and Hank are the software we had access to and could run tests on. The former is a commercial product and the latter is a research tool, but it is not available as free download on the internet. But it is always possible to contact The University of Iowa and their team that work on Hank. It can also be noted that there are many other simulators which could prove to be better. The hard part is finding them. I found a website for the US Department of Transportation – Federal Highway Administration which had a small list of a few simulators for and descriptions. I also found another called Virtual Terrain Project that also have a few links to both vehicle physics and simulators. However, these will not be looked closely at since comparing two simulators is going to take enough time.

4.2 Logical Representation of the worlds

Below follows three types of logical representations for the different simulators that I had the opportunity to evaluate. It should probably be mentioned that the last chapter here is about a proposed standard that I do not think anyone use yet. It is included in case it in the future becomes a global standard as it tries to become. If you want you could say that these are the main parts of the simulators since they restrict what can and what can not be represented in a simulated world. However the main simulator programs do also play an important role in what you actually can do with these logical representations of objects.

4.2.1 Environment Description Framework (EDF)

This is the method that Hank is based on. Below follows a few points that were described in a document consisting of a collection of information from Willemsen (2000).

I quote: “

- Adjacency is defined in terms of which objects are in front of, behind, or next to the nearby autonomous agents
- Curvilinear coordinates naturally represent locations along a road-like surface thus facilitating road and lane tracking behaviors
- Relative spatial locations are easily defined with respect to the curve which is instrumental for following and obstacle avoidance behaviors
- Roads in EDF derive their width from their segment
- In EDF, *range attributes* describe curvilinear expanses of road in which specific behavioral or societal rules apply.
- Similarly, EDF uses *features* to model localized, situated information that exists at specific distances along the length of the road.
- During a simulation, EDF maintains information about the simulation objects located on each road’s surface at any instance of time. This information can be queried providing behaviors with important road occupancy detail. Object location on roads, as well as how those objects’ locations relate to each other, facilitates following and obstacle avoidance behaviors.
- EDF models zone-based, road characteristics as *range attributes*
- A feature is a logical EDF component that models localized, cross-sectional information on the road’s surface
- Traffic control devices that regulate entrance into and movement through the intersection
- Behavior in EDF intersections is regulated by traffic control devices (*e.g.* stop signs, traffic lights, pedestrian walk signals) that control traffic flow on a per-corridor basis. Bidirectional corridors, such as sidewalks, are controlled unilaterally by a single traffic control state.

“

Upon having had the opportunity to try the simulator I discovered that the world logics were divided up into both an EDF-file and an SDL-file. The SDL-file covered the information about positioning and creation of certain objects. The only objects I worked

with was the camera and human driver objects starting position and positioning objects that create and destroy traffic, called Twinsources and Sinks respectively. The EDF-file contained information about intersections and road segments. It ranges from positions and lengths to what they connect with and probability that someone would choose that road. Important details like direction of traffic and type of lanes also lie in this file.

4.2.2 ST Software network files (NET)

STRoadDesign was a bit short on information. This is the information their site had.

- The design of road geometry is highly intuitive and user-friendly.
- Different kinds of lanes are supported, such as exit lanes, bicycle lanes, pavements for pedestrians, complex lanes for weaving on highways etc.
- Different kinds of traffic lights can be added to intersections. The control strategy of these traffic lights can be completely adjusted.
- All kinds of road markings and road signs can be added.

However, in the manual which comes with the product, there is a lot more information. ST Software is a package consisting of a scenario handler, a world designer and the simulator. But it has not focused on trying to follow any special standard (Like the OpenDrive standard that has been suggested.) It does not support a couple of types that the OpenDrive standard suggests, like for example height difference (for hills or tunnels) and neither does it have cubic coordinate system for road shape, but instead just straight or curved sections. It has a lot of positive things speaking for using it that will be listed below (in addition to what has already been listed). ST Software keeps you above the actual coordinate system and let you mostly work with coordinates relative to the road so it is hard to distinguish it from whether it is a cubic coordinate system or not.

- Easily expandable, for example it is easy to add road signs of desired type
- Runs on an average PC, low processing power requirements.
- Graphical representation of the world viewable with OpenSceneGraph
- Structure on logical road representation is readable, if opened with notepad. (A bit hard to read though)
- Straight roads or curved roads.
- Highly customable in settings (like position of a sign in x, y, z dimensions, align buildings along a road section, set number of lanes etc)
- Editor allows copying huge segments of roads, terrain, intersections by just selecting an area and choosing to copy it.
- Supports 3D models, billboards (just a flat texture, good for trees to decrease the sight distance) and differently textured ground.
- Roads have a special ID per lane, and it goes from lane 0 and up (unique numbering).
- Roads have a start and an end node, which connect them to other roads (most often through an intersection).

4.2.3 OpenDrive

OpenDrive is not an actual simulator in itself, but a suggested standardization of the logical representation for a simulator, or more specifically for a road network representation.

- Cubic polynomial description of road shape, which means it has a cubic expression and allow for shapes supported by cubic math.. It does not necessarily mean it is three dimensional, but it can achieve a certain shape on curves.
- Lanes have road marks, speed limits, height
- Also curvilinear coordinates and normal coordinates
- Object lists to keep track of what is on the road
- Intersections have path information and controllers (traffic lights, stop signs)
- XML format for scenario logic.

4.3 Comparison ST Software vs. Hank

There are many differences between working with ST Software's simulator and Hank's simulator. At first impression ST Software was the easy to use simulator and Hank the powerful and flexible simulator. You can see a brief overview of the simulators below.

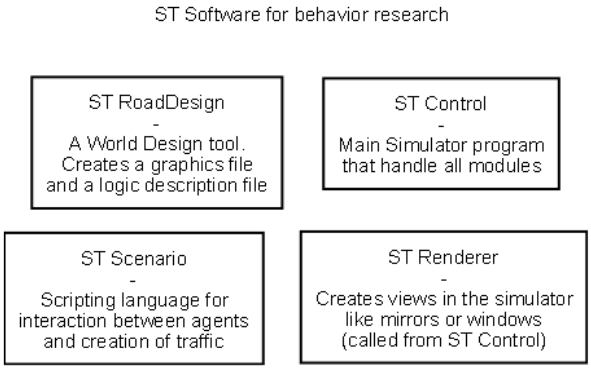

Figure 2 The different parts of the ST Software Simulator

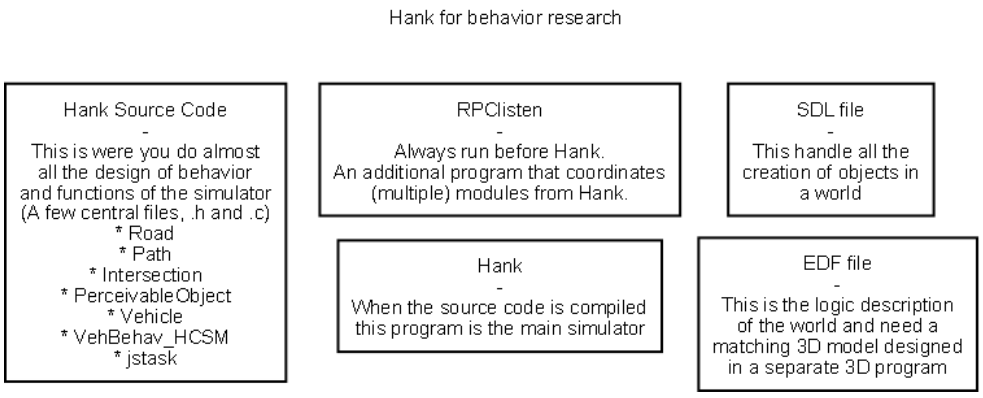
Hank for behavior research
Hank Source Code
-
This is were you do almost all the design of behavior and functions of the simulator (A few central files, .h and .c)
* Road
* Path
* Intersection
* PerceivableObject
* Vehicle
* VehBehav_HCSM
* jstask
RPClisten
-
Always run before Hank.
An additional program that coordinates (multiple) modules from Hank.
SDL file
-
This handle all the creation of objects in a world
Hank
-
When the source code is compiled this program is the main simulator
EDF file
-
This is the logic description of the world and need a matching 3D model designed in a separate 3D program

Figure 3 The different parts of the Hank Simulator

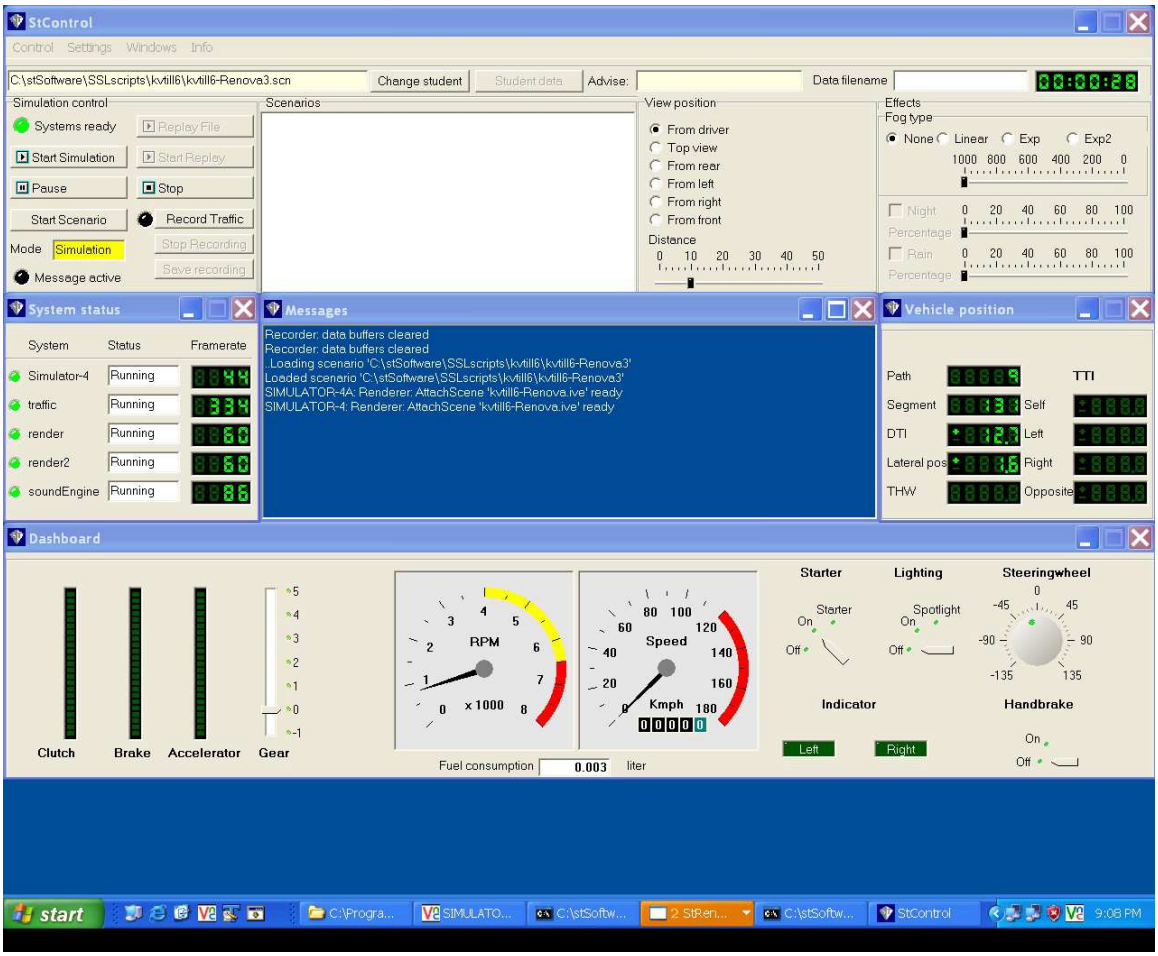

Figure 4 StControl which allows keeping track of a lot of data during simulation

4.3.1 Language

Hank mainly uses the C programming language for implementing behavior and HCSM and EDF scripting for the world logic. ST Software has its own script language called STScenario for behavior that is pretty straight forward and easy to learn, and their world logic is handled by a world editor called STRoadDesign. When it comes to detecting errors Hank has the usual C programming debug tools for the C code, but no way to check for errors in the EDF or SDL files that describe the world and object placements. STScenario is possible to run a syntax checker depending on the text-editor you use. I believe the version that had one of these syntax checkers available was called Textpad.

It is hard to say which language was preferable, since I knew C from before and the scripting language was well documented. The STScenario script was a bit more fool proof together with the syntax check, but Hank offered more freedom.

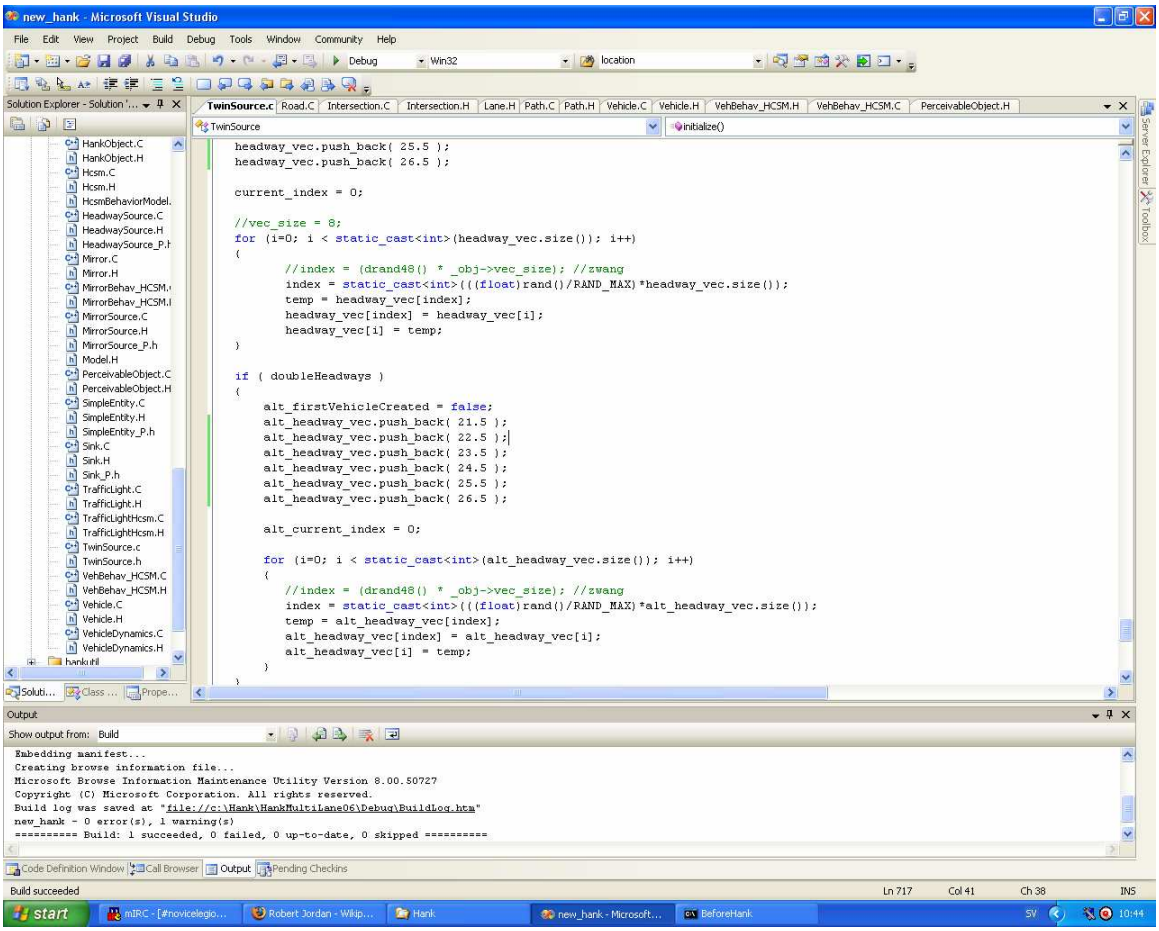

Figure 5 The Hank design tool (Hank source code in C-language)

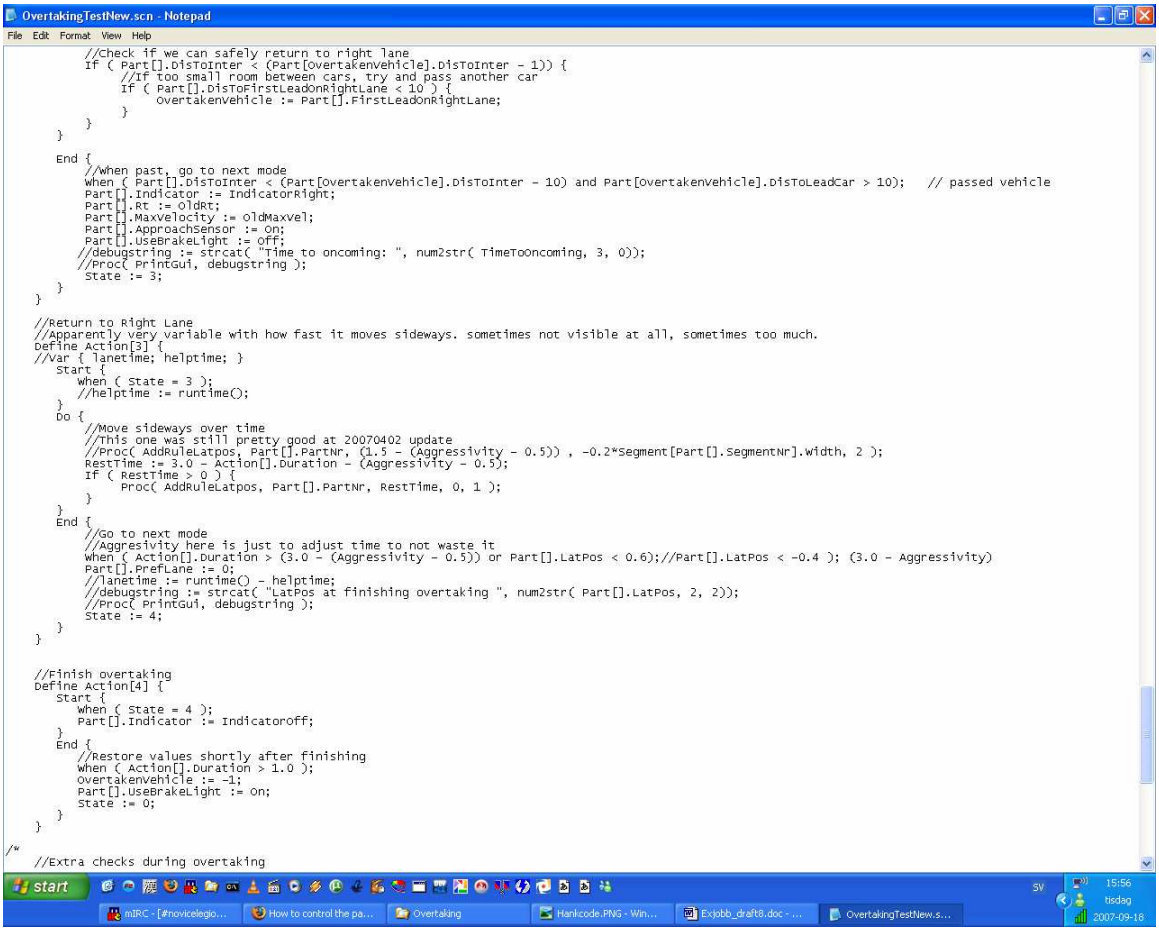
OvertakingTestNew.scn - Notepad
File Edit Format View Help
//Check if we can safely return to right lane
If (Part[].DisToInter < (Part[OvertakenVehicle].DisToInter - 1)) {
//If too small room between cars, try and pass another car
If (Part[].DisToFirstLeadOnRightLane < 10) {
OvertakenVehicle := Part[].FirstLeadOnRightLane;
}
}
}
End {
//When past, go to next mode
When (Part[].DisToInter < (Part[OvertakenVehicle].DisToInter - 10) and Part[OvertakenVehicle].DisToLeadCar > 10); // passed vehicle
Part[].Indicator := IndicatorRight;
Part[].Rt := OldRt;
Part[].MaxVelocity := oldMaxVel;
Part[].ApproachSensor := On;
Part[].UseBrakeLight := Off;
//debugstring := strcat("Time to oncoming: ", num2str(TimeToOncoming, 3, 0));
//Proc(PrintGui, debugstring);
State := 3;
}
}
//Return to Right Lane
//Apparently very variable with how fast it moves sideways. sometimes not visible at all, sometimes too much.
Define Action[3] {
//Var { lanetime; helptime; }
Start {
When (State = 3);
//helptime := runtime();
}
Do {
//Move sideways over time
//This one was still pretty good at 20070402 update
//Proc(AddRuleLatpos, Part[].PartNr, (1.5 - (Aggressivity - 0.5)) , -0.2*Segment[Part[].SegmentNr].width, 2);
RestTime := 3.0 - Action[].Duration - (Aggressivity - 0.5);
If (RestTime > 0) {
Proc(AddRuleLatpos, Part[].PartNr, RestTime, 0, 1);
}
}
End {
//Go to next mode
//Aggresivity here is just to adjust time to not waste it
When (Action[].Duration > (3.0 - (Aggressivity - 0.5)) or Part[].LatPos < 0.6);//Part[].LatPos < -0.4); (3.0 - Aggressivity)
Part[].PrefLane := 0;
//lanetime := runtime() - helptime;
//debugstring := strcat("LatPos at finishing overtaking ", num2str(Part[].LatPos, 2, 2));
//Proc(PrintGui, debugstring);
State := 4;
}
}
//Finish overtaking
Define Action[4] {
Start {
When (State = 4);
Part[].Indicator := IndicatorOff;
}
End {
//Restore values shortly after finishing
When (Action[].Duration > 1.0);
OvertakenVehicle := -1;
Part[].UseBrakeLight := On;
State := 0;
}
}
/*
//Extra checks during overtaking
start
mIRC - [#novicelegio...
How to control the pa...
Overtaking
Hankcode.PNG - Win...
Exjobb_draft8.doc - ...
OvertakingTestNew.s...
SV
15:56
tisdag
2007-09-18

Figure 6 An example on how ST Scenario scripting language only need a text editor to design behavior.

4.3.2 User's Manual

Hank has no user's manual yet, which makes it hard to get started with. I have had two options; Guessing and testing or emailing questions. For fear of messing with the wrong parts of code I usually emailed questions to learn the basics, which is a slow and time consuming way compared to manuals. Hopefully Hank will one day have a user's manual if they want others to be able to quickly learn to use their simulator.

ST Software has a user's manual that is still being written. A few sections have yet to be included, but most if not all of the scripting functions have been documented. What I remember missing was how to add traffic lights to the world in the world editor. It proved to be simple after looking at the editor's menus. Another minor user's manual problem I ran into was for example how the manual tells you to not go above 180 degree turns while the creators of the simulator informed me to not pass 90 degrees to avoid problems since a 180 degree turn had caused problems for me.

In this case I was a lot in favor of ST Software, since working without a manual is difficult, and C code is not always clear to a person who has not written it.

4.3.3 Controls

The Hank version we received did not include any realistic driving vehicle functions. It had a simple bike mode where you could stand still, or quickly reach a preset low speed. By altering the code slightly I managed to add a very simple car behavior in that it has 5 gears and stop if you do not keep the throttle down. There is of course no limit to how realistic you can make it, but I did not feel like I had the time and motivation to try and design controls that functioned just like in ST Software, which would have allowed for a more accurate comparison.

ST Software had a very rich and detailed control system. You had to first turn the key to start the ignition (though this was done with a button), then you had brake, acceleration and clutch. It also had reverse and 5 gears.

So put bluntly ST Software is a lot better than the Hank version we have been allowed to work with, even if it is possible to make both function at equal quality with some extra work.

4.3.4 Worlds

The qualities of the simulated worlds were pretty similar, and both used OpenSceneGraph as base. ST Software has a slight edge over Hank in that they offer an editor so you can with relative ease design the world you wish to test. In the Hank case they do not have a graphical editor yet and do all the work in a 3D-program for graphics and a text editor for the logical descriptions of the roads. This means there is some work involved in manually keeping the logic to fit with the graphics.

For designing worlds ST Software seem to be a much better alternative currently. And since testing agents depend on how accurately you can create the situations you want to test I would say Hank is not that attractive for tests yet.

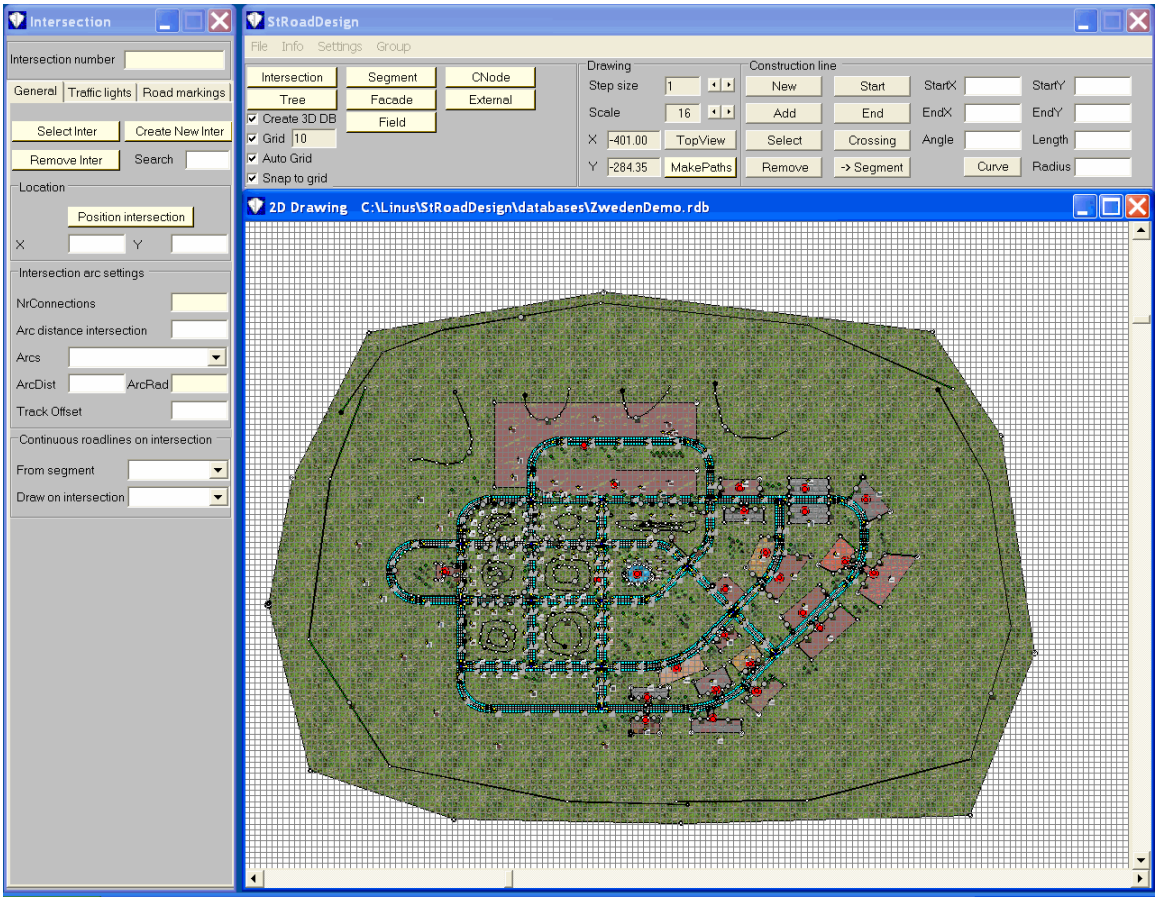

Figure 7 An image displaying the ST RoadDesign tool for creating worlds

4.3.5 Hardware

Our Hank version is currently a slimmed down version running on only one computer and screen. It is possible to increase the number of computers and screens to get a larger field of view. The ST Software hardware that Linköping University has bought is already two computers for each simulator (a total of 4 simulators), with 3 screens each.

So in this case they are both fairly even, although currently ST Software has a slight edge with already having it all set up at Linköping University. I think Hank could provide similar quality if needed.

4.3.6 Agent Designs

This is probably the heaviest part of the comparison since our goal is to work with agents and to create realistic, close-to-human behavior. I have already mentioned that they differ in language, and this is probably where the agent designing issues arise. In the Hank case we have regular C code. This means that if we need certain information, we can find it even if it might require adding more code. This means that there are no limits (except those from what the logical description of the world holds). ST Software on the other

hand uses a scripting language, which limits what information you have access to and what actions you can perform. For example you could not set acceleration, but had to set a new desired speed and let the agent-vehicle accelerate itself. There is the possibility of emailing ST Software and ask for certain changes or additions, but you should probably not expect them to perform all changes you might have a need for.

On the other hand ST Software's clear scripting language and documentation make it a lot easier to work with, making it a good tool for testing ideas. If you look far into the future I would probably say Hank will reach furthest but if you have to test things right now ST Software has more things ready.

4.3.7 Modules

A difference I noticed while working with both simulators was that the ST Software way of having these user-made agent behaviors in a separate scenario-file meant that it was easy to keep them free of each other and not worry about them interfering with each other. And if you actually would want them together it would be possible to adjust them and simply include them in another file as you usually use modules. This would allow you to easily remove for example overtaking if you would not want it to occur.

Hank on the other hand required you to work in multiple files even when just designing one agent. This meant that by default there is no easy way of using modular design, but since it is done in source-code there should be a possibility of writing your own support for modules.

At the current progress of Hank it seems like ST Software is a user-friendlier alternative if you want to add and remove behaviors after designing them.

4.3.8 Overview

While working with Hank and ST Software it became clear that ST Software had an advantage of being easily overviewed, while Hank was much harder to get a good overview of. While working in Hank I noticed I often had to switch between files and functions to adjust behavior, while in ST Software it was always confined to a single file. After a while you started to lose track of what you originally were working with after tracking some error through 2 different files. Something that seemed to help with this was that I usually printed things where I found a problem so when I had to backtrack after fixing a problem I just had to find a cout-function with the correct text.

So it seemed to be much easier to keep track of code in ST Software than with Hank.

4.4 Conclusion

It seems to me that both have strong points and weak points. ST Software seems to be on the winning side of ease-of-use, while Hank seems to be more difficult to get into and work with but will allow more freedom. In the case of time not being of importance I would suggest Hank to be the best choice, because of its freedom and versatility. However Hank is going to need some time invested before it can reach the level of ST Software and surpass it.

From a university point of view, Hank is probably on research level and ST Software on student course level. In other words I think ST Software could be a valuable tool for AI courses, while it is a bit too restricting for some of the things you might want to perform during research. It should be mentioned that ST Software most likely is a better choice for actual driving tests, since it is much easier to design scenarios with than in Hank. It is just unfortunate that it has certain limits when it comes to the agent designing, like how some basic rules are preprogrammed at low level and inaccessible for the scenario programmer to disable or adjust. . A typical example is how a car detecting a meeting agent instantly decides that it should return to a lane with his own direction of flow.

Figure 8 The main screen during simulation in ST Software's simulator

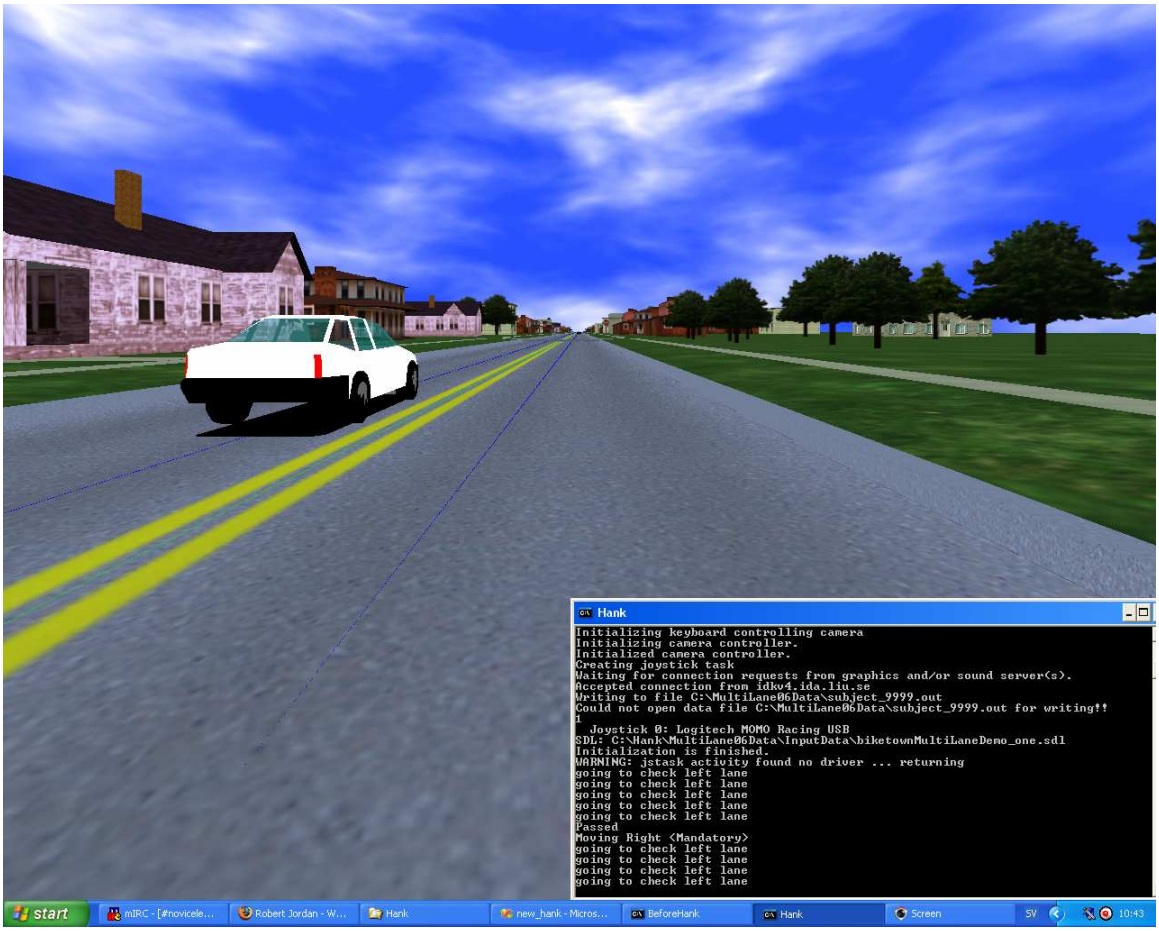

Figure 9 The main screen of Hank during simulation (DOS window is invisible when not debugging)

5 Implementation of simple agents

To evaluate the ST Software and Hank, the decision was made to implement three common agents. These three agents were an overtaking agent, a traffic lights agent and a meeting agent. They started out as simple as possible and were later expanded as time and simulators allowed. Apart from designing these agents there was also a need for designing worlds and populating them with traffic that would interact with the agent.

Something that bothered me during my work was that after all the research on equations I did, the simulators already had built in functions for almost all of it and I ended up mostly using basic math and calculating expected times from distance and speed and then compared these. However I think that keeping it simple also had its benefits. After all it is the same way of calculating the distance and time a human would perform while driving. They would assume the meeting car follows the traffic restrictions and then they would look how far away they are and guess on how much time it would give them.

5.1 Creating a world

The first step would have been to create a world. However in the case of Hank we had no world editor. I asked the team that created Hank if they had any editor available but they said they had not yet made an editor for worlds. They designed their worlds currently using a 3D program like 3D studio MAX and then wrote by hand an EDF logic representation of the world that fitted the 3D world. Instead they offered us a world they had designed themselves consisting of a few intersections along a long road. Later it would show that these intersections were a problem and had to be removed from the EDF logic to allow testing of overtaking agents. Removing the logics meant that the world graphically still had intersections, but the agents within the world only saw a straight road.

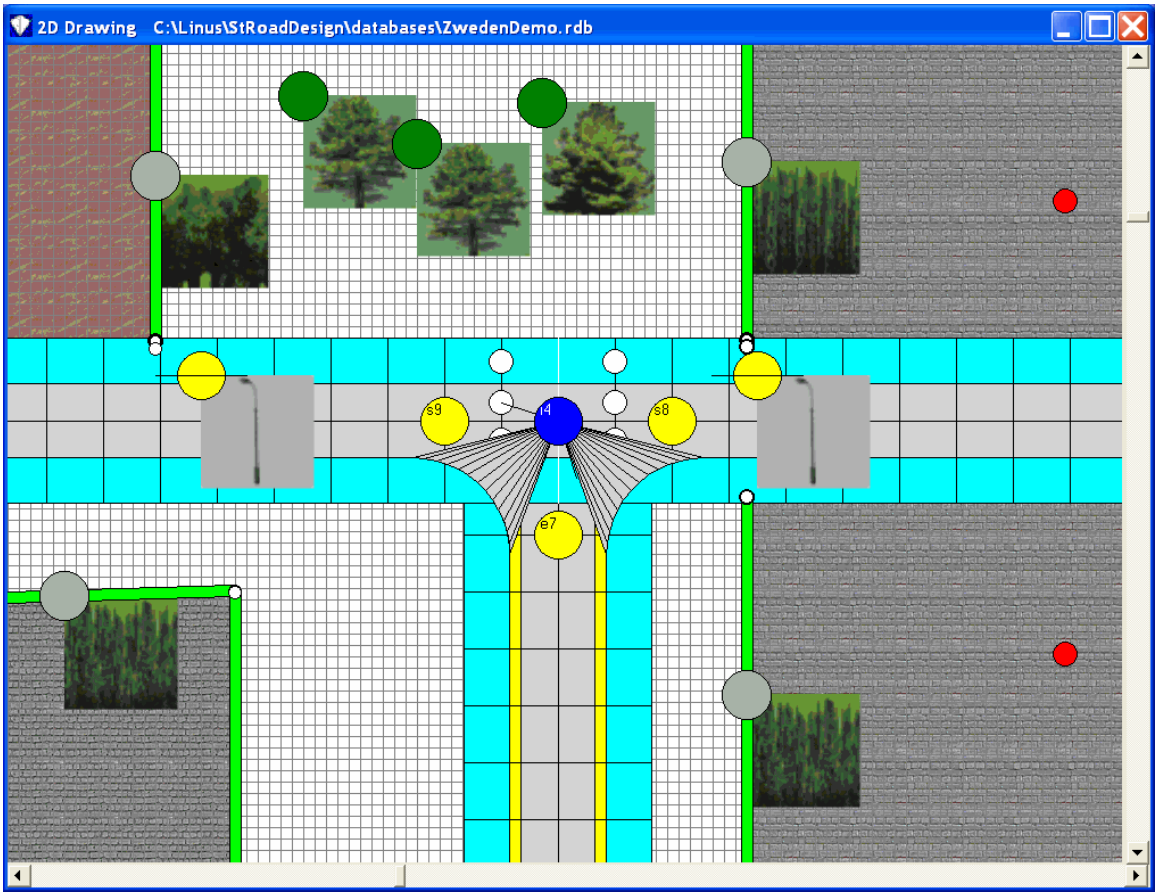

Figure 10 Zoomed in image of an intersection in the ST RoadDesign tool.

ST Software on the other hand offered a program called STRoadDesign that allowed you to from a top down view design your own world. When saved it automatically generated a matching logical representation of the world as a .net file and a graphical file that could be viewed as an OpenSceneGraph-file. Many people might remember the old wooden train tracks you could lay out and then move trains along them. STRoadDesign was not that straight forward. There were many things to observe and to think about.

First off all road segments had a direction. If you started on the west end and drew a road eastwards, the next road had to start at that east end and go somewhere else, or another road segment could be made from somewhere but it had to end at the west end. Combining road segments incorrectly would cause the logic to be incorrect.

Secondly designing curved roads was not as easy as you might have thought. The general method of designing curved road segments was to select curved road type, then set a starting point at a straight roads end, then follow up with if it was left or right bent (looking from the starting end of the curve). When direction was selected, you wanted to choose degrees it would bend and length (in radius). Considering that it is possible to simply click start here, bend right and end here, this method does not seem that obvious at first. In other words it was easy to miss some of these steps. However if you try this simple three step approach you will notice that your world file becomes corrupted. I can

not explain why it is in this way, but certain programming choices that were made during ST Software's design resulted in this. If you want curves you need to put a little extra work into it and do it the right way.

Thirdly it could be useful to know how roads work together with intersections. A road has the same identity over several segments. It is only intersections that cut them off into different IDs. Intersections can be simply straight added at the end of a road and then connected to more roads, or you could cross two roads with each other and get a white square at the crossing part. This white square is possible to change into an intersection by targeting it and inserting an intersection. Leaving a white square alone would not be recommended since it would mean that two roads are intersecting without the corresponding logic saying you are allowed to turn. It would also look like a road is lying on top of another road.

STRoadDesign covers a lot of other desired objects as well. Traffic Lights, while not described in the manual, was straight forward to add. You selected an intersection and went under a tab labeled Traffic Lights. It let you add new ones and position them, and of course group them together. A typical intersection would have 2 by 2 traffic lights, so two traffic lights would be group 1 and two would be group 2. The simulator itself handles the work of synchronizing them with each other.

Other notable objects that I did not use but exist are road signs, billboards (for custom signs or simple obstacles like a flat moose). You also have the ability to customize the amount of lanes and their types on each road.

5.2 Populating the world

To evaluate any kind of agents it is important to first be able to create these agents in the simulation. Hank and ST Software deal with this in different ways. In Hank you have Scenario Description Language (SDL) file with information about places where cars are created and other places where cars are destroyed and an EDF file with general world description. The destructors are called Sinks and are set with a few values: Radius and Location. Radius seems to be a circular zone centered on the Location at which the cars will be destroyed.

Location has a few values itself: “RDL”, name, position, lane and direction.

- RDL is an old variable that probably should be removed from the code because it does not do anything special. Still it is best to add it to the function when calling it.
- Name is the EDF file’s name of a road segment.
- Position is a one-dimensional coordinate on a segment, which tells how far from the beginning of a segment a vehicle is.
- Lane informs which lane the object is on. Usually it is positive (1,2,3) for right lanes and negative (-1,-2,-3) for left lanes according to the segment’s orientation.
- Direction of a lane is simply “pos” or “neg” or “both” which follow the segment’s orientation. “Both” seemed to confuse my cars so I avoided it, I am not certain when it should be used.

The objects who handle creation of cars are called Twinsource and have the ability to create cars at two points; Location and Alt_Location. To my understanding the creation of cars occur with a random delay picked out of a list of delays. The random does however not seem to have a new seed each runs so the pattern of the traffic is identical between runs. This is good for testing but not as well for giving a feeling of realism. Also the creation of cars seems to switch between the normal and alternative location. Additionally the Twinsource has the values speed and proximity. Speed sets the speed of a car on creation. Proximity is a bit more advanced. From the beginning it was designed to only check towards the human driver, and when this driver passed that point, it stopped checking. In other words it gave you a half a circle of proximity checking for human drivers. We chose to change this in the code so it could check in a full circle, to make it easier for our own designs.

<Create cars at two ends of an intersection>
create TwinSource(speed = 25.0, proximity = 360.0, location=sdl.locator("RDL", "West_3rd", 240.0, 1, "neg"), alt_location=sdl.locator("RDL", "East_3rd", 43.0 , -1, "pos"));

<Stop/Remove cars when they have passed (and reached the end)>
create Sink(radius = 30.0, location=sdl.locator("RDL", "East_3rd", 6.0, 1, "pos"));
create Sink(radius = 30.0, location=sdl.locator("RDL", "West_3rd", 280.0, -1, "neg"));

ST Software deals with this in a different way. Everything is handled through the same script language that also creates the agent. There is actually two very different ways within ST Software to create traffic. One method was traffic stream, which is used with the command CreateTrafficStream(). This method did not seem to give much control over the created agents compared to the other method. Among other things, it may be difficult to add participant scenarios to these. A participant scenario is a triggered event or loop that is connected to a certain agent, which means not every agent will be able to trigger this scenario. StopTrafficStream() does not destroy the created cars, it only stop to create new cars.

```
<Create cars with>
a := SetHighTrafficDensity();
CT_MaxVelocity       := 50/3.6;
CT_Velocity   := 50/3.6;
CT_PathNr     := OtherPath;// from right
CT_DisToInter := OtherDisToInter;
CT_RoutePath1        := NextPath;
STREAM1 := CreateTrafficStream();

<Stop with>
a := StopTrafficStream( STREAM1 );
```

The other method required slightly more code, but that was mainly because it allowed for more customization. This method is about creating individual cars until you feel like you have the amount you wanted. With the code PNr = CreatePart(integer) a new participant (or agent as we have called them before) is created. By then going through different Part[PNr]. Using ”attributes” you can set all the things you might need. For example Part[PNr].Path and Part[PNr].Lane to determine where it is placed. The previously mentioned participant scenario is added a bit differently by Proc(Addscenario, PNr, “scenario number”). Proc() is actually a call for system defined procedures. These range from car cabin settings, data storage procedures, road networks to participant, driver or scenario related functions. To really see what they offer there is no option other than to read the manual.

```
<Create cars with>
b               := 1 + rnd(5);
PNr             := CreatePart(b);
If( PNr > 0 ) {
        Part[PNr].RemoveOnDistance
        := 1550;
        Part[PNr].MaxVelocity
        := 50/3.6;
        Part[PNr].Velocity
        := 50/3.6;
        Part[PNr].Route
        := Clear;
        Part[PNr].PathNr
        := 3; //2
        Part[PNr].DisToInter
        := 500;
        Part[PNr].Lane
        := RightLane;
        Part[PNr].RuleOvertaking
        := Off;
        Part[PNr].RuleYellowTrafficLight
        := On;
        Part[PNr].Rt
        := 0.7;
        Part[PNr].Route
        := Straight;
        Part[PNr].Route
        := StoreRoute;
        Proc( AddScenario, PNr, 150 );
}

<Stop with>
Stopped by turning off the car generation
event with a stop condition.
```

5.3 Overtaking agent

The agent for passing maneuvers is the most advanced out of the three agents (Overtaking, Traffic Light and Meeting), but it is also probably the most self sufficient one that only needs to work for itself and not cooperate with any other agents. All it really has to do in its simple form is determine if it should or should not perform a passing maneuver. The actual passing maneuver should be quite simple and follows the following steps:

- Accelerate and change lane
- Pass another agent
- Return to the original lane

While planning it I hoped there would exist a way to write this so it would automatically work the same in a straight road as a curve. After having worked with the simulators I noticed they did work this way which saved me a lot of problems.

My intent was to try a few different equations to see which was best. In the end though I ended up doing it the easiest way by simply checking time left and making the decision based on if the agent actually managed to pass or not during tests. The equations I had had in mind from the start but ended up not using were the following:

Equation 2.10 and Equation 2.11 would probably have been a good first test — copied from earlier parts of this thesis for the reader's convenience.

$$d_1 = 0.278 t_1 (\bar{s}_p - m_{pi} + \frac{\bar{a}_p t_1}{2})$$

$$d_2 = 0.278 t_2 \bar{s}_p$$

Since we also need some references to what kinds of values are appropriate I was going to use selected parts of a table from Jenkins & Rilett (2006).

Table 1 Overtaking Speeds and Distances

Road Speed (km/h)	Impeding car speed	Passing car speed	Distance (m)
30	29	44	200
50	44	59	345
70	59	74	485
90	73	88	615
110	85	100	730
120	90	105	775

The same article also presents another table with an average test results.

Table 2 Average test results for needed distance during overtaking

	50-65 (km/h)	66-80 (km/h)	81-95 (km/h)	96-110 (km/h)
Distance (m)	317	446	583	726
D1	45	66	89	113
D2	145	195	251	314

These values could be useful to compare with during tests.

Other alternatives to designing an intelligent agent for passing maneuvers could have been to use Equation 2.7 or Equation 2.8, which offer both a test on going by distance to decide if we should perform the task or not, and going by time. When going by distance we could have begun with trying similar values to those of previous equations. When going by time we would not have had any good example for yet — Equations copied for easier overview.

$$T_{\min} = 2T + T_b \qquad T \approx \sqrt{3}\frac{W^{3/2}\sqrt{A}}{V^2} + 2.4\sqrt{\frac{W}{A}} \qquad T_b = \frac{(L + L_1)}{(V - V_1)}$$

$$D_{\min} = 2D + D_b \qquad D \approx 2.4V\sqrt{\frac{W}{A}} \qquad D_b = V\frac{(L + L_1)}{(V - V_1)}$$

A table from the same article (Shamir 2004) might give some helpful insight into values for these equations.

Table 3 Values from using advanced equations for overtaking

V (m/s)	W (m)	A (m/s^2)	D* (m)	T* (s)	V1 (m/s)
15	3	3	36	2.47	12
25	3	4	52	2.1	15
25	4	2	84.96	3.43	20
35	3.5	4	78.67	2.26	20

It is unclear which D and T the star-marked items refer to but instantiating with some values should probably show which formula they fit best into.

The above equations cover the equations which were focused on passing maneuver. The equations below are not designed for overtaking but might work with or without some slight adjustments. The equation known as Equation 2.1 (Shladover & Tan 2006) together with a rough estimate on how long it takes to pass the agent in front could be a way to design a simple agent. — Equation copied here for convenience.

$$T = \frac{(y - x)}{(\dot{y} - \dot{x})}$$

First of all, the directions would be different so if we called the agent driving in the opposite direction *x*, then we would have had to give it a negative speed value or rewrite the equation to contain addition in the denominator instead of subtraction. Which would have been the easiest would probably have been shown when looking closer at how the agents' speeds were acquired. After having worked with it I would say that in Hank the speeds were a vector so you could have used vector math but at the same time simply

converting it to a regular integer or float value sounded easier and is what I did in the end. ST Software on the other hand had float values to begin with and no vectors.
But TTC would not be enough to determine if the agent should try a passing maneuver or not. We also needed to estimate how long it would take to pass the car in front. My proposal at first was to determine time to overtake depending on acceleration, passing speed and to use a simple calculation about distance needed to travel in that speed and see what time the result would be. Roughly something like this:

$$T_{overtake} = \frac{C}{v_{own} - v_{ahead}} \qquad \textbf{Equation 5.1}$$

The speed *v* with indexes *own* and *ahead* should be self explanatory. *C* would be a number relating to how long the cars are combined, and then some added distance behind and in front of the agent. It should probably be the distance required to travel if the agent ahead was standing still. Then the check to see if we should let the agent overtake the other agent would be given by.

$$T_{available} - T > 0 \qquad \textbf{Equation 5.2}$$

5.4 Traffic Light Agent

The simple traffic light agent was assumed to be easier than the overtaking agent. It involved a bit different logic. It could not switch from green to yellow to green without seeming suspicious for the driver, so the cycles needed to remain, but the duration of the cycles could be changed when the driver is far away from the light, so that it would give the correct signal when the driver arrives. I assumed constant cycle time in the system, even if it is better for traffic if the lights adapt to traffic it would keep the planning of the agent simple.

So the basic idea was to first determine how long it would take for the driver to reach the traffic light. This should have worked fine with Equation 2.1 since it supports setting *x,* which would be the traffic light, to zero without problems. This equation is assuming that people keep a fairly constant speed and there are no noticeable obstacles on the way to the traffic light. After determining the time to arrival, we would have to adjust the cycle time.

Now the question was how the traffic light cycle works. A Swedish document of a chapter 8 called “Projekteringshandlingar för trafiksignaler” by Swedish Road Administration (2002) has some interesting figures.

$$R = S_u / V_u - S_f / V_f - R_g - (G - T) \qquad \textbf{Equation 5.3}$$

This equation calculates how long time the light needs to be red while other cars have green or yellow in the other direction at a four way crossing. It is natural when you think about how the purpose of a traffic light is to avoid collisions. *R* is time in seconds that it

has to be red. *S* index *u* is the distance a car needs to travel inside the intersection (and its length), *V* index *u* is its speed. In the same manner *S* index *f* is the distance a car would need to drive to intersect the other cars trajectory. *S* is a car that has to stop for the red light. *V* index *f* is its speed. (Which is confusing since the car would most likely stop at the red light, maybe this is incase the car is currently closing in on the intersection. Anyway, division with zero seems unfriendly). *R* index *g* is red-yellow time. *G* is yellow time and *T* is green time. But since we are focusing on simple agents at first we will most likely avoid using this equation to begin with.

Swedish Road Administration (2002) has a lot of figures and some of the figures in the document are a bit hard to read. However the one below was informative and clear.

Table 4 Typical Swedish Traffic Light Durations

Zebra crossing go	12 s, ends with blinking green
Green min time	5 s
Red-yellow	1 s (assumed value out of image)
Yellow time	5 s
Order	Red –> Red-Yellow -> Green -> Yellow -> Repeat

As we can see there is no minimum red time given, which is because the red time depends on the grouped traffic lights none-red times added together. Assuming a simple 4 direction intersection without any special lanes for turning left it should be 2 traffic light cycles that need to fit with each other. When one of them is green, red-yellow or yellow, another has to be red.

Now that we have a simple system idea for the traffic light cycle, we need a way to adjust the cycle time to adapt to an incoming car. From the numbers above we could probably derive a minimum cycle time and an average cycle time. Then we would have some possibilities to adjust the cycle to be somewhere between those.

What is needed to be done next is determine what signal on the traffic light we want when the car arrives, and how long it will take the car to reach the traffic light. Going the most basic and simple way first, we can assume no obstacles on the way and rather constant speed. (Equation 2.1)

It is also possible to use the equations that use distance instead of time, if we calculated the time from the speed and distance. There is not any real equation for distance which I have found, but it should be possible to extract from the road information.

5.5 Meeting Agent

The last agent is an agent designed to meet up at a special meeting location with the driver. The most important thing here is timing, and there is not a lot of room for error.

The tricky part here is determining if one should choose a certain agent and then adjust for it to meet up at the right time, or if one should choose a random agent, which seems like a good candidate for reaching the point on time.

Both seem to have their own strength. Choosing a certain agent is fast, but might require it to act strangely to arrive at the determined time. It might have to drive really slow, or speed through several intersections to reach the point in time. However since we are looking at beginning with a simple agent this might be the best choice. The other option is to try and evaluate a large amount of vehicles to determine if an agent is at a fitting distance to arrive on time. This means a lot of computing effort to select the car, but for the remaining part of arriving on time it will not need much alterations in speed and behavior.

The meeting agent is most likely very similar to the traffic light agent, except there is no need for adjusting a cycle to make it look realistic.

6 Agents structures in ST Software

6.1 Overtaking Agent

Data found in a car object:

- ApprCar – gives *ID* of the first approaching car on the same road.
  - Most likely the one coming from opposite direction, since there exists a *LeadCar* which give *ID* of the car ahead of us in the same lane and *RearCar* which give *ID* of the car behind us.
  - There also exist a related variable named *DisToApprCar* for distance and also *FirstApprOnMyLane* if one is performing the overtaking and is on the other lane or *FirstApprOnLeftLane* before overtaking. As with *ApprCar* there also exist versions for distance starting with "*DisTo*".
- Rt – reaction time is available, which could prove useful for testing.
- PrefLatPos – Something for the more advanced functions. This determines if the car should be positioned at the center, to the left or to the right in the lane. Other things for the more advanced cases involve distance information to bus stops, zebra crossing, etc.
- LatPos – this one gives the lateral position of a car, might be useful for overtaking agents if we do not want to use a lane changing command.
- Other information – There is also information about, for example, max speed, current speed, participant scenarios and much more.

From what I can tell about the documentation it seems that it should be possible to do the experiments on this platform, assuming the variables work as I interpret them. However I have my worries that more advanced type of algorithms might be hard to make. For example most checks are for closest object, which might make it hard to time two cars to meet at an intersection, if there are two intersections between both cars.

6.2 Traffic Light Agent

Data found in traffic light object (actually more of a path object):

- Traffic Light (just color information)
- GreenPhase, YellowPhase, YellowRedPhase (time it stays a certain color)

Data found in a car object, related to traffic lights and intersections:

- Velocity
- DisToInter – The distance to next intersection
- RuleRedTrafficLight – If the car should stop for red light.
- RuleYellowTrafficLight – If the car should stop for yellow light.

That should be the most vital information we can read when trying to make traffic lights adjust its color to a desired choice for a simulation.

6.3 Meeting Agent

Because of the way STScenario works it would be inaccurate to actually call this an agent. It would be much more of a scenario which picks the existing most fitting car based on distance to an intersection or distance to human driver. And after choosing, it would have to adjust its speed to look realistic and yet still meet at the desired point. The only problem I was worried about before implementing it was how to adjust the surrounding traffic to not be in the way of the meeting agent, since those would obviously interfere with the calculated time for meeting. After implementing it I noticed that it was not as a big problem as I thought.

7 Agents structures in Hank

7.1 Notes about Hank

It took a bit longer to receive the Hank simulator than ST Software to the university, and since Hank did not have any real information about details in the form of a user's manual the focus on designing the structures for Hank was left until later. This meant that the Hank structures are influenced by the work that was done on ST Software's simulator and have a slightly different look.

Instead of a focus on which variables and functions that Hank might have, I have written down the implementation strategies I used in ST Software to design the agents. The general idea was to use the same structure and order of the ST Software code but adjust for the differences between the simulators. So in the Overtaking Agent's case we wanted to use the same kind of state machine as in ST Software.

7.2 Overtaking agent

There exists a pre-defined function for changing lanes which should in theory work perfectly for overtaking. The only real difference between changing a lane and starting an overtaking is how you deal with the information about the other lane, so in theory we should be able to add a special case for when the flow of the traffic is towards the other direction.

So in theory the Hank Overtaking structure would be about 4 steps.

Step 1
- Check the Left Lane's direction of flow.

Step 2
- Change Lane (with special rules if the flow was opposite direction)

Step 3
- Increase Speed (since during overtaking you do not want to stay on this lane for long)

Step 4
- Change Lane back (this should work with the already existing version)

The code would be mostly in Path.C and VehBehav_HCSM.C in this case.

7.3 Traffic Light Agent

Due to time constraints this part was neither thoroughly planned nor attempted to be implemented. An educated guess would be that you would work in the files VehBehav_HCSM.C and TrafficLightHcsm.C. However I did not have time to ask much about how they function so I do not have any suggestions for how to implement it.

It should be noted that back on page 9 I discussed EDF represented Traffic Lights Willemsen (2000). It is possible that Hank already had a good Traffic Light system that in some special way could be informed of which color to display on arrival. The question would mostly be where you would set this color since there was nothing to input information from during runtime.

7.4 Meeting Agent

While this was not implemented either, a bit of the work behind planning was put into it incase time would have allowed for implementation. Equation 2.1 is enough to make a simple meeting agent. We could have needed to create an object type called “goal” or “meeting point” from which we would have determined time until meeting with the driver car. This time would then have been used in combination with the meeting car to determine speed from how far it has left to travel in that time.

Step 1.

- Calculate time until arrival using Equation 2.1

Step 2.

- Decide on a car that will meet up.
    - Either pick one at random
    - Or try and select the car that would logically reach the point at the same time given their expected arrival time because of distance and road speed.

Step 3

- Adjust the speed of the meeting car to make it reach the point at decided time.

Step 4.

- When almost the full time has gone and the driver is closing in on the meeting point, it might be a good idea to recheck how well the meeting agent is doing. Might need to adjust the speed a bit to reach the point at a more exact time.

8 Adding a feeling of realism

There are several reasons for why we would like as realistic behavior as possible when dealing with agents. These agents are more likely to exhibit interesting, intelligent behavior, and would therefore create more enjoyable simulated environments. Successful implementation of human-like behavior can also support various theories on human behavior. Finally, human-like agents would elicit a more natural behavior of human experimental participants. If the simulation feels too artificial people might not react to the situations as they would in the real world. For example we could design a passing maneuver agent that drives exactly how a human *should* drive, but it would feel weird because most humans tend to drive unsafe.

Of course there is a great amount of factors that determine how people drive. Age, sex, peer pressure, fog, snow, stress, Advanced Driver Assistance Systems (ADAS), drinking, type of car, etc. We are going to try to narrow these down to a few adjustable variables that we will be able to set to simulate certain behaviors. In order to keep it simple we will continue to look at passing behavior and traffic light behavior.

8.1 Traffic light behavior

It is probably easiest to start with traffic light behavior. There are two special behaviors that is not rule-following or safe, which is how agents usually are designed. Both are unsafe, but only one of them involves breaking a rule.

The rule breaking one is driving against red light (and at least in Sweden, also yellow light). This can happen for two reasons. The more common one would be because of lack of attention and the other would be because the driver thinks it is safe to drive because he can not see any dangers and is in a hurry. With these cases it sounds like it could be a good idea to have a value for attention and a value for how likely the agent would ignore rules because of stress, we could call the latter a stress value.

The other case is how a human driver approaches a traffic light. If they are closing in on a green light with a speed of 50 km/h they tend to keep the speed, instead of slightly slowing down to adjust for the possibility of the light changing. And if it changes to yellow a human driver tends to do a quick choice between increasing speed or breaking hard if close to the traffic light. The stress and attention values should work here also.

Both of these cases can be read about in an empirical study by Liu (2007). Something interesting for behavior creation that Liu mentions is how people react to yellow light (referred to as amber in the article). The drivers start with slowing down while deciding whether to brake or speed by. Another interesting thing in the study was the tables and graphs describing different information. The relative speed for example was reported to be from 11 km/h below the speed limit up to 23 km/h above, with the more common speeds being 11 km/h above for men and 1-2 km/h above for women. The last interesting thing to mention from Liu's report is the mean speed for seeing a green signal. Suburban

50 km/h road was reported to have a mean of 40.5 km/h, while the urban test with 40 km/h had 34 km/h. These numbers are interesting because if you would design a normal and rule abiding agent they would always treat yellow as something to stop at, while in reality people actually speed by it on average. It also plants a seed about if agents perhaps should have a variable telling them to drive like a woman or like a man since behavior seemed to be dependent on sex. Of course the difference in behavior in urban and suburban surroundings also sounds like something that would add flavor to an agent.

On its own it does not sound like such a huge problem with people driving past green traffic lights at high speeds, since it makes the traffic flow faster. But the problems start to appear when we look at other typical driving behaviors like how close to the car ahead people drive. Broughton et al. (2007) have done a study on the headway values that most people keep while driving, which are in general lower than what would be called safe. Driving manuals frequently suggest 2-3 seconds headway while driving but Broughton et al. (2007) mention that Wasielewski (1979) stated that the average in fact was more around 1.36 seconds. Another number that pops up in the article is how an experiment showed headways between 0.55 and 2.2 seconds at 65 km/h. These numbers could turn out to be helpful in designing a realistic agent.

Sometimes drivers adjust their driving style to the environment, such as fog. But our intent is to leave those cases out of the agent for the time being, since most tests would not involve fog, but Broughton et al. (2007) list a lot of information about these cases.

8.2 Passing behavior

Passing behavior and to a larger extent driving on roads with several lanes or rural roads that allow passing have a lot of different behaviors that a realistic agent would need to adjust towards. An article by Wills at al. (2006) lists a lot of answers to a questionnaire about common driving behavior. I will list some of the more important points for building a realistic behaving agent for simulation purposes. Since it is a questionnaire the actual percentages are likely to be higher because to know that you missed a stop sign you have to notice it afterwards.

- Fail to notice pedestrians crossing when turning 67%
- Miss “stop” signs or “give way” signs 60%
- Underestimate the speed of an oncoming vehicle 57%
- Attempt to overtake a car signaling to turn right 47%
- Become angered by another driver and give chase with the intention to give them a piece of your mind 26%
- Find yourself distracted by other thoughts 23%
- Disregard speed limit on freeway 63%
- Drive close to the car ahead to signal to them to drive faster or get out of the way 62%
- Race away from traffic lights with the intention of beating the driver next to you 51%
- Become impatient with a driver on the left and overtake on the right lane 48%
- Become angered and indicate hostility in some way 46%
- Disregard speed limit on residential road 43%
- Sound horn when annoyed 41%
- Stay in a lane that merge with the other lane ahead just to force yourself into that lane when it ends in the last minute 37%

Since it was not reported how often these things occur we are lacking an important fact to design good agents, but it gives us an idea of what kind of scenarios the agents should be able to perform, depending on if we want them to do specific driving simulation events, law-abiding driving or “human” driving. It would have been nice if you could use these percentages as a hint on how many cars should act a certain way in a scenario, but the problem is that the questionnaire only asked if they had ever done it and not how often they do. It could be that some people do it daily and other people only once a year or maybe they even have stopped doing so.

Bar-Gera et al. (2005) have made other interesting observations about passing behavior. They mention that even while some passing maneuvers seem rational there are also passing maneuvers done because of the driver’s competitive side. While it is obvious that a car will pass a car driving significantly slower, Bar-Gera et al. discovered that even if the car ahead of the driver kept a speed 3.2 km/h above the driver’s own speed, the tendency to pass was over 66%. It was also reported that the drivers kept a rather constant speed as long as the road was clear, but if they noticed a car fairly close ahead of them, they accelerated with intent on passing, even if the car ahead drove above the speed limit. These moments when a car catches up to faster speeding cars are of course closely linked to the human behavior of not being able to keep a constant speed but sometimes losing speed and sometimes driving faster than intended.

The results of these tests also showed that people have a tendency to pass cars that vary their speed too much. If they approach a car they instantly assume that car is slower (which it obviously is) and that it will stay slower for the rest of the trip. This means a driver who can’t keep their speed constant risk getting overtaken more often than one

who can keep it near constant. The mental load of not passing could be the reason why people prefer passing even if they need to raise their speeds considerably because following a car requires a constant adaptation of that car's speed.

These studies would seem to give the impression that a good cruise control in a car would lower the amount of passing, assuming it can adapt well to different slopes on the road.

Another bit more general source for unsafe driving is how a driver deals with cooperation and competition. Vanderhaegen et al. (2006) mention how cooperation generates positive interferences and solve the negative ones as fast as possible, while competition focus on the negative without trying to minimize them. Some good examples they mention are:

- Cooperation – Driver stops to let a person out from a stop sign road.
- Competition – Not helping a signaling car to change lane to your lane but just ignoring it.

Additional situations I could think of myself are the following:

- The "gear" system at entry lanes in heavy traffic. Cooperative drivers let every second car enter the road, while competitive drivers may ignore the cars trying to merge.

Many of these things are pretty general and would take too much time to implement them all, so the focus will remain on overtaking agents, traffic lights and meeting, and from there we will see what behavior adjustments that can be made.

8.3 Suggested variables

From the information found there was a lot of things that could be added but to keep it simple I planned on keeping the amount of variables low. Just a few simple variables to adjust behavior for the agents should be enough if designed with care. While there is for example carefully, competitively, clumsily, etc. behavior, these can probably be controlled by a few underlying cognitive variables.

An approach I tested was a variable for aggressiveness of the overtaking agent in ST Software. The value ranges from 0 to 1, with 0.5 as the number for your average driver. Setting aggressiveness to 1 would make the agent keep shorter distance and change lanes sharper which also means faster. It would also overtake at less distance to meeting cars than your average agent. Similarly, setting it smaller would make the agent drive extra safe and keep larger distance and change lane slowly and not overtake unless meeting traffic is far away. However some fine tuning would be needed to make it work as detailed as desired. Currently it makes a difference, but it is hard to tell if it is different enough.

It would have been nice to implement a stress variable that would affect the aggressive variable and also a new variable about awareness about surroundings. But it is currently hard to treat the surroundings in a good way. A human sees the graphics and trees and houses can cover your field of view, but the logical database does not include information

like if a car is concealed or visible. You could probably make a simple version where the agent just randomly ignores red-lights or stop sign. (However I did not spot these as easily turned off in the manual of ST Software so it could be trickier than I thought.)

It would be preferred if all the previously mentioned behavior details could be implemented, but for now it will have to do with a simplified version only affecting a few details.

9 Results

9.1 Results using ST Software

Below follows the three types of agents I designed in ST Software and a description of the results I got. Since the results are more focused on how it feels while sitting at the simulator it is hard to transfer this feeling into words. You should preferably sit by the simulators yourself to see the results, but I have done my best to describe the problems and what worked well to give an idea of how the work went.

Figure 11 A photo of one of the four stations of ST Software's simulator at Linköping University.

9.1.1 Overtaking

Overtaking was actually supported from the beginning with ST Software, but in the first version we were limited to overtaking without approaching traffic. But since it is always interesting to be able to adjust things we decided to try making our own version to compare with the built in overtaking algorithm and hopefully get more adjustability than the built in version. After some questions regarding limitations of ST Software we received an update that improved the overtaking agent that was built in, but it still felt a

bit unnatural. It had a habit of starting and overtaking at very dumb moments and having to abort overtaking to avoid a collision.

Example of what is required for the built in overtaking system:
Part[].RuleOvertaking = True;

Our own agent on the other hand managed to perform much better for the actual beginning and ending of an overtake, but I could not get past a limit by the default rules that made our overtaking car act like a coward and stop mid-overtaking and try and return to the previous lane. Because of this behavior being a very basic, default behavior it would have been difficult to change it. The solution, which had its own limits, was to remove the agent's forward checking during an overtaking. This lead to not being able to adapt to changed circumstances, since the agents were blind when it came to meeting cars during overtaking so they might drive straight through them. This would happen if the meeting cars did not act as the overtaking agent expected them to act when it calculated if it was safe to pass.

Here are a few of the key functions in the code:

```
Do {
  //Check for cars getting close that we can overtake
  LeadVehicle := Part[].FirstLeadOnMyLane;
  LeadIsClose := False;
  If ( LeadVehicle >= 0 ) {
     If ( Part[].DisToFirstLeadOnMyLane < 30 and Aggressivity > 0.0 ) {
       LeadIsClose := True;
     }
  SecondCarDistance := Part[LeadVehicle].DisToLeadCar;
  }
```

This code checks for vehicles ahead of the current agent. If it would happen to be within a distance of 30 meters and the agent is not told to be aggressive but very careful then this code should inform the agent that it is getting close to a lead car. An extra check for the distance to the car ahead of the lead car is added to make the agent handle passing more than one car better.

Even with the extensive research in behavior and suggested formulas the knowledge collected beforehand was not particularly useful when implementing overtaking in ST Software. ST Software offered a lot of built-in functions for reading various data—other data, outside of this, was not accessible. As a result, the final version of the overtaking agent used a sort of state machine for overtaking. The overtaking state machine has 5 states. Begin overtaking, change lane left, pass, return to right lane, and finish. The basic idea is that each of these are confined to a type called action which work just as member functions inside a *PartScen[]* which is a custom scenario attached to a participant or in other words agent. Each action has a starting case and a finishing case. So in general all these states in the state machine are triggered by a state being a certain value and finishes by setting the state to the consecutive value.

Examples:

```
Start {
        When ( State = 2);
        //Accelerate to pass faster
        Part[].MaxVelocity := OldMaxVel + 10*Aggressivity;
    }

End {
        //When past, go to next mode
        When ( Part[].DisToInter < (Part[OvertakenVehicle].DisToInter - 10) and
Part[OvertakenVehicle].DisToLeadCar > 10);   // passed vehicle
        Part[].Indicator := IndicatorRight;
           Part[].Rt := OldRt;
           Part[].MaxVelocity := OldMaxVel;
        Part[].ApproachSensor := On;
        Part[].UseBrakeLight := Off;
       //debugstring := strcat( "Time to oncoming: ", num2str( TimeToOncoming, 3, 0));
       //Proc( PrintGui, debugstring );
        State := 3;
    }
```

This code part is the starting and ending condition and events of the passing state. As you can see there is a *When* case that determine what triggers the *action*, and in the clauses there we have something that is done once at start. In the end part we have the same type of *When* case that determines when this action stops and a few variables that are set once at ending. You might notice the commented *debugstring* variable which is very useful for printing to the console of ST Control during runtime. To actually print it you have to call the *Proc(PrintGui, debugstring);*.

Now to resume a few of the key functions of the overtaking code:

```
Do {
//Check for when we don't have meeting cars closeby
      TestVehicle := Part[].FirstApprOnLeftLane;
      If ( TestVehicle >= 0 ) {
         SumVelocity := Part[].Velocity + Part[TestVehicle].Velocity;
         If ( SumVelocity > 1.0 ) {
                TestTime := Part[].DisToFirstApprOnLeftLane/SumVelocity;
             }
            // If we are going to need to pass 2 cars, simply assume shorter time until
meeting.
            If ( SecondCarDistance < 20 ) {
                  TestTime := TestTime - 3;
            }
      }
      If ( TestTime > (10 + 2*(0.5 - Aggressivity))
             and Part[Part[].RearCar].Indicator = IndicatorOff
         and Part[Part[].LeadCar].Indicator = IndicatorOff
         and Part[].FirstRearOnLeftLane = Absent) {
            Part[].Indicator := IndicatorLeft;
            OvertakenVehicle := LeadVehicle;
            Part[].Rt := 0.3;  // decrease preferred headway to leadvehicle
            OvertakingTime := runtime();
         State := 1; // start overtaking
       }
}
```

This is the decision part of the code that determines if the agent would believe it could do an overtaking without colliding with meeting traffic. The *Do* means that it keeps running every cycle after having been started by a *Start* condition. As you can see I decided to cheat on this agent and let it use the meeting traffic's actual speed. Any real human would have to guess the speed (and usually assume that the meeting car follows the speed limit) and make their choices after considering how much time this would leave them. As you can see here I used one of the most basic equations (Equation 2.2) to determine how long time it would take until collision. It is slightly different in that I add both speeds and then divide with distance between the two cars, but it returns the expected time until a car would have driven that distance if the other car was static.

I ended up experimenting with the numbers to get overtaking to happen, instead of translating the distance formulas suggested earlier into time instead of distance. Table 1 under 70 km/h gives us the distance ~500 m which translated into safe overtaking time would be 13 seconds (assuming those numbers counted on the sum of the speeds to be 140 km/h). However my experimenting showed that about 10 seconds was enough (but could end up resulting in a close re-entrance into the lane in if it actually started at around 10 seconds and not with larger window of time.). In the current version of the overtaking

agent that has a variable for how aggressively the agent should drive the safe distance can end up anywhere between 8 and 12 seconds. If the TTC is above this number, the agent will try to overtake.

Some other important parts of the code were the part that speeded up lane changing. There was a very simply way of changing lanes in this simulator that I considered was a bit too slow and I wanted more control over the speed. Another also important part is that if a car behind or ahead of the overtaking agent signals for a lane change, the agent does not try to overtake.

Example of default method:
Part[].PrefLane = 1;

PrefLane works in such a way that 0 means the rightmost lane and then higher numbers step to the left in lanes. The name stands for Preferred Lane and tells the agent which lane it should try and drive in. I mentioned this problem earlier about how we had a problem because there is a very deep and basic rule in the simulator that says you should not drive in a lane with meeting traffic. I disabled this with the function below.

Example of how to turn it off:
Part[].ApproachSensor := Off;

Example of the custom lane changing function:

```
Do {
        //Move sideways over time
      RestTime := 3.0 - Action[].Duration - (Aggressivity - 0.5);
        If ( RestTime > 0 ) {
            Proc( AddRuleLatpos, Part[].PartNr, RestTime,
0.5*Segment[Part[].SegmentNr].Width, 1 );
      }
   }
```

As you can see I integrated *aggressivity* into this *Do* loop to make aggressive behavior mean a faster lane change. But the main function of this loop is that it calls a procedure called *AddRuleLatpos* with the information about which agent, time, distance and priority. By changing the *RestTime* value it is meant to give a fast movement at first that slows down more and more the closer to the lane you get so that it avoids going too far and ends up in a shrinking sinus shape. In other words it would drive past the center of the lane then try and return to it and approach it more and more in a wavy pattern. From testing it seemed like it did not wave much at all with poor values but it did move too far and end up outside the road before it sneaked back to the road.

The next thing we needed to implement was of course to tell when the agent could return to the right lane again. It was done with the following code.

```
//Check if we can safely return to right lane
If ( Part[].DisToInter < (Part[OvertakenVehicle].DisToInter - 1)) {
 //If too small room between cars, try and pass another car
    If ( Part[].DisToFirstLeadOnRightLane < 10 ) {
     OvertakenVehicle := Part[].FirstLeadOnRightLane;
 }
}
```

Here we begin with checking if we are closer to the next intersection than the agent we planned on overtaking. The reason we want to be closer is that then we know that *FirstLead* calls will find a different agent than the one we were passing. If we are closer we can run a check on *DisToFirstLeadOnRightLane* which will tell us the space between our overtaken car and his lead car. Too short room and we change our overtake target to the lead car and pass him as well. A problem with this is that you could if unlucky end up at a traffic jam and need to pass 10 or more cars because there is no room to turn back to the original lane. In a real scenario like this you could probably use your indicator and have people make room for you to change lane back. However our agents are not that advanced yet and I decided to for the time being let this problem remain since it is not a common problem.

The finishing part of this passing state tells a bit more about when you actually are considered to be past all the overtaken cars.

```
When ( Part[].DisToInter < (Part[OvertakenVehicle].DisToInter - 10) and
Part[OvertakenVehicle].DisToLeadCar > 10);   // passed vehicle
```

Comparing this to the loop we ran during passing we look for 10 meters in free space for returning and we want to be 10 meters ahead of an overtaken car. I have not seen any data about safe values for this decision and when I am driving I usually go by the rule that I should see the other car in the rear mirror. But it did look good in the simulator and it is usually the important part when working with simulators.

I did happen to add a little extra action that covered a special scenario to make the agent safer.

```
Define Action[6] {
    // Check if he car infront of us also did an overtaking, and adjust overtaken car
    Start {
       When ( State = 2 and Part[].FirstLeadOnMyLane = OvertakenVehicle );
       //Proc( PrintGui, "I better change who I'm overtaking!" );
          OvertakenVehicle := Part[].FirstLeadOnRightLane;
    }
    End {
       When ( Action[].Duration > 1.0 );
    }
```

This code was hard to test because it is not a common problem. It should in theory cover if the car ahead of you changes lane while you start to overtake them. This should not happen to begin with because of a check before the agent start an overtaking that makes sure no car behind or ahead is signaling. But as we all know people can forget signaling and then this action can help avoid problems. An example of when this could come in handy would be when they are passing a human driver and said driver also start to pass, because human drivers do not follow the same rules the agents do.

Something I chose to not include was checking if there is room to return to the lane after overtaking was finished. Instead I have a simple system that lowers the TTC by about 3 seconds if there is reason to suspect we need to pass two cars. There are no actual facts behind this idea, but just something I tried. The reason for subtracting time is that the decision to pass is taken if TTC is above a certain value and I could then use the same decision in both cases since if we expect it to take longer we would receive a shorter available time. The alternative would have been to increase the value for considering it to be safe to overtake from 10 to maybe 13. The situation rarely happened during testing so I am not certain it works perfectly. A few advanced ideas could not be implemented easily so I left them out of the implementation. Sight for example seemed to be too hard to evaluate for the agents so I doubted I would be able to tell if they had a bush or house in the way of other agents. Unexpected changes to the traffic like accidents or road construction work were also too hard to take into consideration, not to mention that these events were not available to be inserted into the scenarios. I did not include the idea of watching for traffic signs or intersections either, but I believe it could be done.

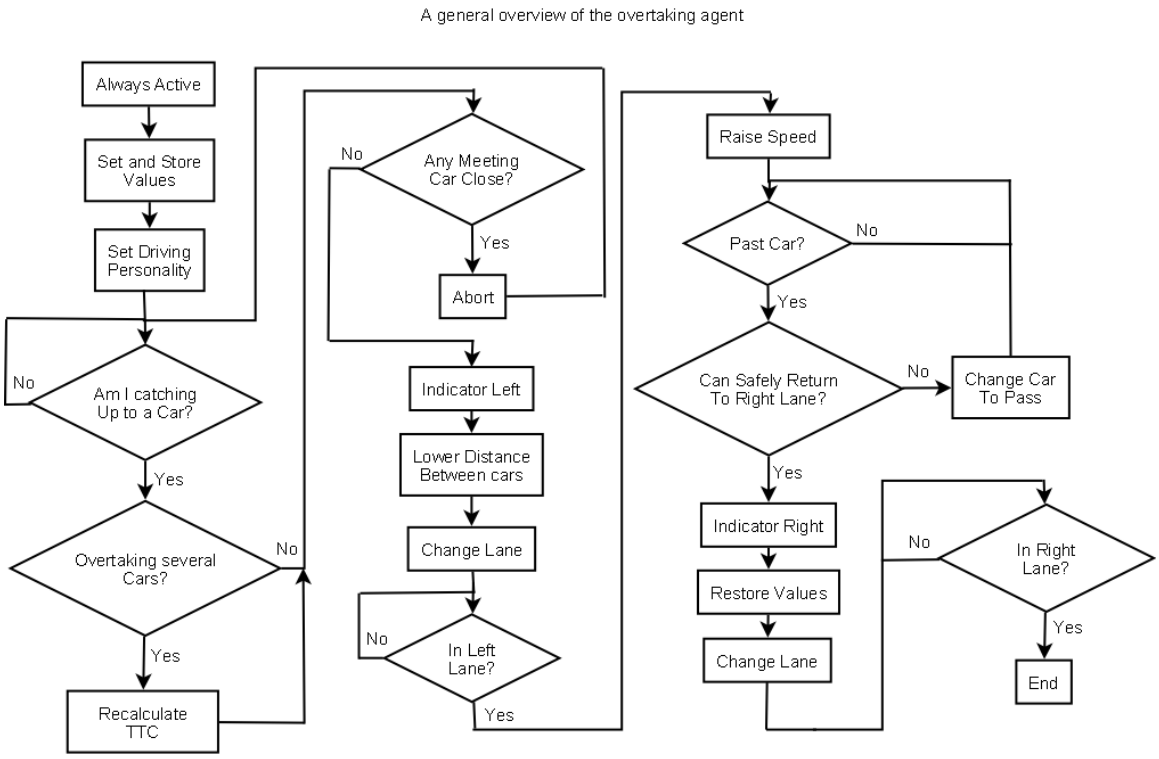

Figure 12 An overview of the scheme for the ST Software based overtaking agent

9.1.2 Overtaking Behavior

Regarding the different types of overtaking behavior, the limits set by the simulator did not leave much room for behavior adjustment during overtaking. However some simpler adjustments before and after was possible. For example there could be a car that did not use indicators when passing (not implemented but a question about setting a variable to true or false). We also allow an aggressive variable to be set between 0 and 1 to adjust the behavior a bit. Value 1 would mean aggressive style, keeping a shorter distance to the car ahead of them, and changing lane sharper as well as overtaking when there is a shorter distance to meeting traffic.

A lot of the situations in the list earlier taken from Wills at al. (2006) (See page 377) did not get implemented. For example the sounding of horn, overtaking on the right lane, becoming angry and giving a chase or underestimating the speed of approaching cars. They were not top priority and would not add that much to the actual overtaking. There is also a difference between the term aggressive driving and actually breaking the law on purpose. I had thoughts about underestimating the speed of approaching cars, but first I wanted them to manage to overtake safely. It should not be too hard if one would want some underestimated attempts. Usually most of these routines are set at creation of a car and knowing which car to set to underestimate distance might be hard since you can never be quite sure how the meeting traffic will be coming. (Assuming you randomize it to make it have variable distances). However it should be possible to add a manual button that the person leading experiments could push to change the behavior of the closest car behind them.

9.1.3 Traffic Light

It seems that to construct an adaptive traffic light for traffic scenarios without making the light change in an unexpected way we have to first know what we want to accomplish. The goal with this agent was to be able to make a driver arrive at a traffic light and know exactly which color it will be. This means roughly three to four scripts setting it at red, green, or yellow from red or yellow from green. It would be possible to do for agents also, but is not supported now because it felt like a more interesting thing for human drivers. The agent uses the distance and speed information from a car and tries to adjust the different lights' cycle time to make the car reach the traffic light at a specific color. A few things that posed a problem here were measuring the distance across intersections and expected arrival time through other traffic lights.

Earlier listed time has been 5 seconds green, 5 seconds yellow, 1 second red-yellow, and the remaining time red, on average. So a cycle red -> yellow -> green -> yellow -> red-yellow -> red would end up as something like X seconds red -> 5 seconds yellow -> 5 seconds green +Y seconds extended green -> 5 seconds yellow -> 1 second red-yellow -> restart, where we let X be the total sum of other colors (5+5+Y+5+1 = 16+Y seconds) to compensate for a 4 directional intersection (we pretend there are no separate traffic lights for turning.) and Y would be an extension in green time to adjust for how many cars need to pass. If there were many cars from south, but few from west, Y would be longer for south than west. (South and north are linked, and so is west and east.)

As with everything the ideas had to be adjusted to the tools in the end. ST Software used a traffic light structure of 10 seconds green light, 3 yellow and 0 seconds red-yellow. So instead of changing these values we kept them. To change it we would only have had to set three Phase-variables in the traffic light object. For some reason I could not get the red-yellow to show in the simulator so I skipped that part and it also was disabled as default with a value of 0 seconds duration so we could then keep everything at default. However this means the traffic light differ slightly from a normal Swedish traffic light.

Instead of a straight extension of the green light time as originally planned I decided to try and keep the proportions between green, yellow and red by giving green ¾ of the time and yellow ¼ of the time remaining to get to the intersection. This meant that setting the light to red and adjusting the values, the driver reaches the traffic light at green. Reaching the traffic light at yellow and red seemed a bit harder to figure out how to set the times. Part of the problem was that a cycle of colors were linked so red was directly related to yellow and green time, and you also wanted the ratio of time to be relatively correct. A 7 seconds yellow and 3 seconds green could have felt very odd since it could mean reaching an intersection where no cars were driving. I decided to just try an idea of setting the active traffic light color to other colors than red, which I had used as a default starting color each time I changed cycles, and see what color the driver had when he reached the traffic light. Luck had it that this method worked fine. Each time I had to adapt the traffic light I just scaled one cycle to the total remaining time to reach the traffic light and then changed the current active color to a color that would lead to the right color when arriving at the light.

Since this method of adjusting the traffic light involves setting the traffic light to a certain color, I also had to experiment with how far away from a traffic light it is possible to actually see the color displayed. During my many experiments it seemed like I grew better and better at seeing the color at a distance because my first impression was that 100 meters in the simulator was good, but later it turned more towards 150 meters, and with time even 150 meters felt like I still could catch a glimpse of the actual color.

Here are some examples of the most important parts of the code:

```
Define Scen[200] {
        Start {
                //When selected in the simulator
                When ( Scen[].Commanded = True );
                //Name the scenario that will be selectable
                Scen[].Description := "Arrive at Green";
                //Only keep it alive for a short moment since it is just a setting scenario
                Scen[].Duration := 2;
                //Set a global variable
                LightScenario := 1;
        }
        End {
        Proc( PrintGui, "Scen 200 deactivated");
        }
}
```

This is how you typically would connect a scenario to the ST Control interface so you can trigger it whenever you want during simulation (though a later restriction on it triggering at a certain distance would force you to start this scenario before this happens). The *Commanded* part places it in the ST Control. The *Description* determines what the scenario is called. *Duration* sets how long it is active until it automatically ends. *LightScenario* in this case is my own variable to determine which scenario we are running. It will result in red, yellow, green, broken or blinking yellow.

```
If ( LightScenario = 1 ) {
        Path[OriginPath].TrafficLight := Red;
        Path[OriginPath].GreenPhase := 3*TimeLeft/4-1;
        Path[OriginPath].YellowPhase := TimeLeft/4;
        Path[OriginPath].YellowRedPhase := 0;
}

If ( LightScenario = 2 ) {
        Path[OriginPath].TrafficLight := Green;
        Path[OriginPath].GreenPhase := 3*TimeLeft/4-1;
        Path[OriginPath].YellowPhase := TimeLeft/4+1;
        Path[OriginPath].YellowRedPhase := 0;
}

If ( LightScenario = 3 ) {
        Path[OriginPath].TrafficLight := Yellow;
        Path[OriginPath].GreenPhase := 3*TimeLeft/4-1;
        Path[OriginPath].YellowPhase := TimeLeft/4+1;
        Path[OriginPath].YellowRedPhase := 0;
}
```

These are the three color scenarios. As you can see I spread the remaining time equally in every case. Green is three times as long as yellow phase. The difference is which color I set it to start at and also a slight adjustment to the yellow phase in the later scenarios to make the desired color last a bit longer and not change too fast after the human driver reaches the traffic light. By testing it appeared to be that if I put it to start on red it will be green when you reach it. Green starting value turned the light yellow when you reached it. Yellow ended up giving you red. Since it worked well I left it this way.

Some advanced parts that did not get implemented were that this traffic light adjustment can not handle multiple intersections or signs that change the expected results, like speed signs. This is mostly because I do not know how to scan a road's different signs between two points, and because it would be harder to calculate the cycle times with varying speeds. It also can not handle multiple traffic lights on the way to the traffic light we desire to change. This mostly because it automatically choose the nearest traffic light for the changes, and it would also be difficult to count on how other traffic lights effect arrival time.

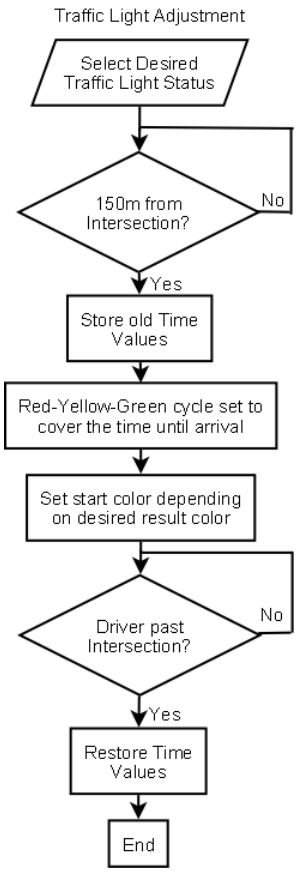

Figure 13 An overview of the scheme for the traffic light agent in ST Software

The results were that my implemented method of adjusting the traffic light for an approaching driver would work fine with the normal speeds 30 km/h, 50 km/h and 70 km/h. The lower the better, since higher speed meant shorter duration of the cycle and I would find that some cycles at high speeds felt too short. You could possibly adjust so that 150 meters is the minimum distance but at higher speeds it is triggered at a larger distance, however at the time of testing I did not think about that.
I also included some extra settings for blinking yellow and broken traffic lights (the whole set, not only a single light), which really just were a setting on the traffic lights. Your option for setting a light was *Red*, *YellowRed*, *Yellow*, *Green*, *YellowFlash* and *Blank*. However this implementation of light adaptation to an approaching driver may not work well for multiple human drivers. It should not be a problem if we adjusted it only for the closest driver and then did not allow any later drivers to affect the traffic light. This is to prevent people from actually noticing the setting of the light, and if we think about it setting the traffic light can only be adjusted for one driver in any case. You can not have a single intersection turning green for one car and red for another at nearly the same time independently of each other since it is the same intersection.

9.1.4 Traffic Light Behavior

Drivers tend to usually drive predictably at traffic lights since it is a very strict rule system. This meant that there was not much that could be adjusted to make it look more real. However I could think of three out of the ordinary behaviors that might happen around traffic lights, and I attempted to implement one of them. The three were how drivers behave when stressed at yellow, unobservant at green, or are behind another unobservant driver at green. The former was attempted in ST Software, but it turned out to be difficult getting agent behavior to actually seem different because there was no way of effecting acceleration except to tell the cars there is a new max velocity. So in the end it was no real visible difference between the behavior of my agent and an agent that simply drives against yellow, simply because the speed difference is not noticeable.

The code looked like this:

```
Define PartScen[150] {
        Var { LimitAction; }
        Start {
                //When we are closing in on a traffic light, start
                When ( Path[Part[].PathNr].TrafficLight != Absent and Part[].DisToInter
< 50 );
                LimitAction := 0;
        }
        End {
                //When we have passed the traffic light, end
                When ( Part[].PathNr = Path[Part[].PathNr].PathToRight or Part[].PathNr
= Path[Part[].PathNr].PathToLeft or Part[].PathNr = Path[Part[].PathNr].PathToAhead );
                        Part[].RuleYellowTrafficLight := On;
                        Part[].RuleMaxVelocity := On;
                        //Part[].Velocity := Part[].Velocity-10;
                        Part[].MaxVelocity := Part[].MaxVelocity-10;
        }

        Define Action[0] {
                Start {
                        When ( Path[Part[].PathNr].TrafficLight = Yellow and
LimitAction < 10);
                        Part[].RuleYellowTrafficLight := Off;
                        Part[].RuleMaxVelocity := Off;
                        //Part[].Velocity := Part[].Velocity+1;
                        Part[].MaxVelocity := Part[].MaxVelocity+10;
                        LimitAction := LimitAction+10;
                }
        }
}
```

As most ST Software code when picking good variable names it is self explaining. If there is an intersection with traffic lights and we are less than 50 meters from it we start this scenario. In this scenario we have an action that runs if the traffic light is yellow and my restricting value of *LimitAction* is below 10. If this is true then it ignores the stop at yellow rule and boost up by about 1 km/h each cycle. If you look closely though I have adjusted it to add 10 each cycle so it in reality only runs once. This means it will be hardly noticeable when it happens except that it drives past yellow. When I did add only 1 each cycle for 10 cycles I managed to create cars that passed the speed of light. It should be mentioned that I picked *MaxVelocity* in this situation to make the agent themselves accelerate, something that ST Software does not allow you to set a value for. If I had gone with just *Velocity* it would have been infinit acceleration in an instant, which I thought might not be a good thing to have.

I considered this attempt at intelligent behavior to have been a failure and not close to how I wanted it to act at all. I would have preferred if I managed to make the agent speed up near a yellow traffic light to simulate how a human would speed up to not have to stop and wait.

9.1.5 Meeting

I also made an attempt at adding a script for cars that meet with the human driver at an intersection, where the meeting cars come in from the right. The results were surprisingly good and more often than not there was a car approaching from the right behind the window frame, making it hard to spot. Something that probably is very exciting to be able to test on drivers.

ST Software was pretty ideal for setting up this kind of scenario, since all the interesting data were easy to access. One lucky thing was that changing speed was all that was really required to change the time of arrival for a car, because one of the things missing in ST Software is the ability to set acceleration. Time to meeting is calculated from distance and speed which are later used in Equation 2.2, and if the T value is too far from zero the agents speed is adjusted so both the driver and the agent has near the same estimated time of arrival. The agent chosen is the current optimal agent. This means it is the agent with the closest TTC compared to the human driver to reach desired meeting point, but it most likely will need to adapt its speed to receive a TTC as close as possible to the human driver. If new candidates appear that could be better, the script changes agent. This prevents for example traffic stocking behind a car adapting its speed to slower than the other cars.

Some flaws of the program were when the agents did a poor timing of switching cars, which could happen at intersections if there was a queue in the way. It also seemed like one would like to not be able to see the road it travels on since the behavior is not what one would call casual. If you were to see the cars using this function from a greater distance you might see how a car slows down compared to everyone else, and then when the car behind him catches up he suddenly resumes the more ideal speed of the road but this other car slows down. The opposite may also happen and is not as big of a problem.

A car catches up to a slower car, and this car then speeds up while the rear car slows down. The reason behind these odd behaviors is that the ideal candidate is picked out of the possible candidates along the road, but intersections can lead new cars into this road that might be better alternatives. (Like a car that does not have to slow down to 10 km/h to not reach the intersection too early)

I am going to list a few parts of the code to better explain how it works.

```
//Am I the meeting car? Check
Define PartScen[250] {
	Var { TimeToInter; Value;}
	Start {
		When ( Part[].PathNr = CarRightPath and Part[].DumVar1 = 0 and
Scen[200].Duration > 1 and Scen[200].Ended = False);
		//What is my expected time?
		TimeToInter := Part[].DisToInterCenter / Part[].Velocity;
		//Am I the best candidate?
		Proc( PrintGui, "Am I a good candidate?");
		If ( abs(PartMainCarTime - TimeToInter) < abs(PartMainCarTime -
BestPartCandTime))
		{
			BestPartCandTime := TimeToInter;
			Part[].DumVar0 := 1;
			// Have we had a best candidate?
			If (BestCarCandId > 0)
			{
				//Clear old best candidate
				Part[BestCarCandId].DumVar0 := 0;
				Proc( PrintGui, "Candidate unset");
			}
			//Set new best candidate
			BestCarCandId := Part[].PartNr;
			Proc( PrintGui, "Candidate set");
		}
		Else
		{
			Part[].DumVar0 := 0;
		}
		//Mark as checked car
		Part[].DumVar1 := 1;
	}
	End {
		When ( Part[].DumVar1 > 0 or Scen[200].Ended = True);
	}
}
```

Remember how participant scenario is a local scenario attached to an agent? Well, this scenario only triggers for agents traveling on the road to the right of the closest intersection. It also is dependent on *Scen[200]* which is the starting scenario which sets up a couple of useful information like expected time to arrive at an intersection for the human driver. *DumVar1* is a restricting value that makes sure each agent on the road to the right of the upcoming intersection is only checked once. This scenario checks for the most fitting agent dependent on expected time to arrive to the center of the intersection. As you can see we perform a comparison between the absolute value of the human driver's expected time minus the current agent's expected time compared to the absolute value of the human driver's expected time minus the current best candidate's expected time. If this agent is better than the best candidate, he is the best candidate. Best Candidates receive *DumVar0 = 1*, while every checked candidate receives *DumVar1 = 1*. Of course when we set a new agent to best candidate we also unmark the old best candidate by setting their *DumVar0 = 0*. If we have checked this agent the action ends. This part of code ended up too long to fit on a single page.

```
//Adjust the speed of the best candidate
Define PartScen[251] {
	Start {
		When ( Scen[200].Duration > 2 and Part[].PartNr = BestCarCandId and
Scen[200].Ended = False);
		//The main car is possibly changing speed, adjust
		Proc( PrintGui, "Best Candidate Probably Found");
	}
	Do {
		If ( TimeChanged = 1 )
		{
			//We need to adjust our speed to match the other car's
			If ( (Part[].DisToInter / Part[].Velocity) > (PartMainCarTime + 1) )
			{
				Part[].MaxVelocity := Part[].MaxVelocity + 1/3.6;
			}
			If ( (Part[].DisToInter / Part[].Velocity) < (PartMainCarTime - 1) )
			{
				Part[].MaxVelocity := Part[].MaxVelocity - 1/3.6;
			}
		}
		//Alternatively switch candidate

		//If candidate catch up to slower car


		If ( Part[].DisToLeadCar < 10 and Part[].DisToInterCenter > 50)
		{
			Part[Part[].LeadCar].DumVar0 := 1;
			Part[].DumVar0 := 0;
```

```
                BestCarCandId := Part[Part[].LeadCar].PartNr;
                Part[].MaxVelocity := 50/3.6;
        }


        //If candidate get caught up by faster car


        If ( Part[].DisToRearCar < 10 and Part[].DisToInterCenter > 50)
        {
                Part[Part[].RearCar].DumVar0 := 1;
                Part[].DumVar0 := 0;
                BestCarCandId := Part[Part[].RearCar].PartNr;
                Part[].MaxVelocity := 50/3.6;
        }

    }
    End {
        // Stop when scenario is over, or another car is a better choice
        When ( Scen[200].Ended = True or Part[].PartNr != BestCarCandId);
        Part[].MaxVelocity := 50/3.6;
        Proc( PrintGui, "Speed Adjustment ended");
    }
}
```

This second participant scenario adjusts both speed and handles switching of best candidates if a new one would appear. (They can appear either by passing an intersection into the road of interest or simply created through the simulator as new agents). As you can see it is only run by the participant (or agent) with the ID identical to the best candidate. If we notice that our human driver has changed his expected arrival time, we change the agent's speed also to adapt to this change. Notice the 1/3.6 numbers at the speeds. It is because the simulator treats the speed as meters/second and to convert it to km/h we need this adjustment.

The candidate switching is dependent on two cases. Case one is if a car catches up from behind and case two is if the candidate catches up to another car. This is slightly restricted so that it does not happen too close to the actual intersection since it would be a very short time to adjust speeds for the new candidate. It could also be a queue in the traffic and it would probably not do much to switch candidate in a queue at the intersection. The whole scenario ends when the agent either stops being the best candidate or the original *Scen[200]* has ended. (Which it does when the human driver is about 15 meters from the intersection)

Some tests were performed just to see how close to a collision the driver got running this script and not stopping for traffic coming from the right. On the following page we have a table of a few of the results, which do not always show the actual feeling you get inside

the simulator. Usually the agents appeared at fitting time to make a normal driver stop for them, with the exceptions of problems with other traffic.

Table 5 Test results on difference at arrival time for meeting agent

Driver km/h	20	30	40	50	60
Driver time to intersection	3.5	3	2	2	1.5
Agent time to intersection	8	6.5	3	2	2.5

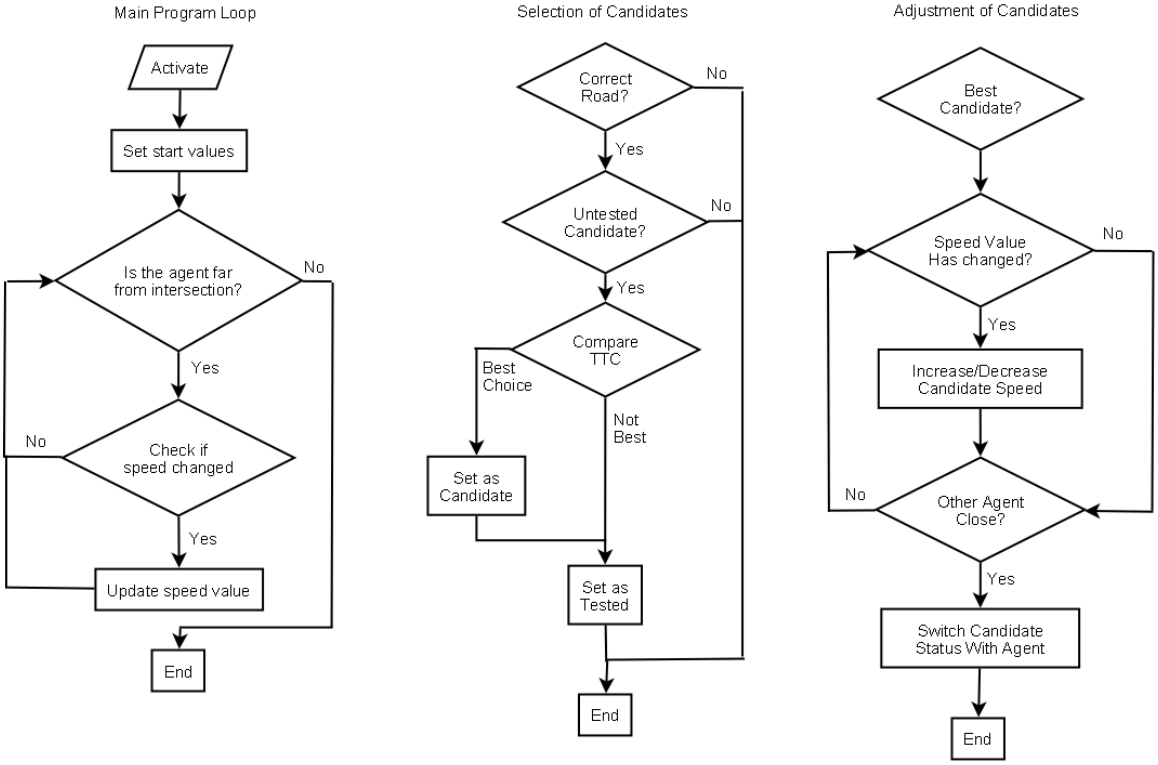

Figure 14 Three smaller schemes describing the meeting agent in ST Software as well as how it is selected and sometimes adjusted

9.2 Results using Hank

This section is going to describe the results of the work with Hank. Just like in the previous chapter about the results using ST Software the results in Hank is best seen when running the simulator. Still images have a hard time capturing the feeling of how agents move so I do not have any images of the tests I performed. However I will try and describe what worked well and what did not work well. Because of shortage of time, I chose to begin with the passing agent. One reason for this was that there already existed a lane changing state machine which I figured I could expand on and use as a reference for designing a passing state machine.

Figure 15 The much simpler workstation for experiments with Hank.

9.2.1 Overtaking - Background

Designing the overtaking agent in Hank seemed to be easy at first glance. There existed a lane changing state machine already that performed the following three things:

- Check if we should consider changing lane.

This one checked if there was any reason to try and change lane. Valid reasons were the roads ending (two roads merge), the current lane does not lead to our destination (separate lanes for forward and turning) or if the agent ahead was driving slower than the agent checking for lane change.

- Pick a mode (*mandatory, limited, discretionary* and *no lane change*)

No lane change is when we actually do not want to change lane, which is the most common one. *Mandatory* happens specifically at merging roads and similar occasions. *Limited* I am not certain but I think it means we have a specific choice for which lane to change to and happens usually near an intersection. *Discretionary* is the other cases when you just want to pass a car driving too slowly.

- Pick a state (*start, waitforgap, moveover*)

This next step is dependent on which mode we received earlier. *Start* is the neutral state which keeps checking which lane would be a good choice to change to. *Waitforgap* is a more active state that looks at the previously decided lane for space to perform the changing of lane. If there is no space, it keeps looking but if there is a space it changes state to the final state *moveover*. *Moveover* is the final state of the state machine and performs the movement between lanes. It is designed as a PD (Proportional and Derivative) controller. Hopefully you have heard of PD controllers before, but if you have not I will try and shortly describe how this one works.

It starts with changing the position in sideways with a lot, but depending on the remaining distance it moves in shorter distances until it has moved as far as it intended to move. It is fine tuned to work well with default settings. If you would desire a faster lane change you could change the PD controller's values. There are two types of problems you can encounter when using PD controllers. If you make it too fast and not a good enough compensation for distance you can get the PD controller to end up as a shrinking sinus wave. This means the car drives too far to for example left, then too far to right, then left, right, until after a certain amount of swings end up at desired location. The opposite would just be that it changes lane very slowly but never pass the desired location. In my own opinion the PD controller seemed to have good behavior by default. The problems I saw on lane changes were most likely related to something else. I will go into more detail about this problem later.

9.2.2 Files that were central in Hank for agents

Let us start with listing the files I worked with to design the agent. A few of the files were not directly related to designing an agent but had to be worked with to adjust the simulator to function closer to how we wanted it to function. All files should be considered as both .c and .h even if I might not have had to change in both.

- Jstask
- Road
- Path
- PerceivableObject
- Twinsource
- VehBehav_HCSM
- Vehicle

Jstask – This file handles how joystick input is treated by the Hank simulator. I had to adjust this to accept inputs from our steering wheel because it used different axis than what our steering wheel used.

Road – The reason I had to work with this file at all was due to a bug in the program. Path has a function called *queryAllLeadersOnRoads()* that called different *queryLeader()* in Road, and one of these were missing an increment of a variable and ended up filling all queries with the first found car.

Path – I would consider this to be the most important file when working with behavior. It is not the file that does the behavior, but it is the file which lets you work with paths along the roads.
A few of the most important functions I worked with in it was:

queryAllLeadersOnRoads(); which checked distance and ID of cars in a certain direction from you (ahead, behind, lane to left and behind, lane to right and ahead etc.) The scan starts from an agent that asks for the query, so the closest car will be the first in the list, if it finds any at all.

gapAvailableLaneChanging(); which returns true if there is space in the lane we want to change to.

computeTargetChangingLane(); which calculates which lane we should change to.

considerLaneChanging(); which returns which mode (mentioned earlier) of lane changing the agent should use.

computeAdjacentLanes(); which returns which lanes we have around the agent. Max two lanes and usually only one lane. (By default it could return no lanes at all since it did not consider the meeting traffic's lane to be valid, but I changed this when making the passing agent)

PerceivableObject – This file is the base of all objects. Sometimes you have to put some variables in this file to keep them available to all types of queries.

VehBehav_HCSM – This file was the second most important one. This one included the state machine for changing lanes (and other things like state machine for passing through an intersection or turning at it).

Vehicle – This is a subclass of percievableObject class and has a few more variables than its parent class.

9.2.3 The implementation of an Overtaking agent in detail

Path.C
considerLaneChanging()

In this function I started with adding an extra check incase it was a limited road structure with only two lanes. If there was a lane to the left that had the type vehicle lane, but not to the right, then the left would be our lane of choice. Same goes for the opposite. The agent would always pick the lane it was not driving on. If these special scenarios did not happen we simply let the default system select lane because we then had an extra lane in our own direction that we could choose. Of course it could have been two meeting lanes and only a single for the agent in question, but I assumed symmetrical roads with either single lanes or dual lanes.

Example:

```
if(leftLane->getInstanceType() == Lane::VEHICLE_LANE &&
rightLane->getInstanceType() != Lane::VEHICLE_LANE)
      {
            nearestLane = leftLane->lane_id();
            nearestRouteConnectionLane=nearestLane;
            laneDistance = abs(leftLane->lane_id()-rpe->lane_id)-1;
      }
```

Next there was a check for mandatory mode. This was by default if there was a lane to change to, but our current lane was ending very soon. I added the extra condition that if my own state machine had the state of *LANERETURN* and an extra limiting variable was set to zero, then it would also become mandatory mode. This means that every time my state machine considered the car to be on a meeting traffic lane, it would try and change lane back as soon as possible again. Something that you can not see in the code is that *nearestLane* is a local variable and *nearestRouteConnectionLane* is a global variable within the path object, which is why we work with both.

Example:

```
if( nearestLane!=0 && aheadRoadDistance <= 0 || (v1->LIU_overtake ==
Vehicle::LANERETURN && v1->LIU_limit == 0))
```

I separate the normal lane changing and my custom built parts in general by different forms of looking at my state machine.

Example:

```
if( v1->LIU_mode )
```

LIU_mode is true when an agent discovers a need for overtaking. When it is true we deal with some information in a different way than usual. For example we do no longer care about *flowdirection()* function's value of the selected lane because we have already determined which lanes we need to use.

checkPassMode()

This function I added myself to check the nearest lanes. This function determines if we should start the overtaking state machine or run the old lane changing state machine, depending on lane choices.

Here is an example of the code for checking left lane. (The code for right lane is similar but for a few changes like – instead of + on the offset.

```
if(_path_elements[pathElement]->direction()==POSITIVE_DIR)
      {
            roadOffset    = rpe->road_ptr->findLane(rpe->lane_id)-
>centerlineOffset()
                  +pathLocation.offset;
            currentLane_id= rpe->road_ptr->offsetToLaneId(roadOffset);
            rightLane     = rpe->road_ptr-
>nextRightLane(currentLane_id);
            leftLane      = rpe->road_ptr-
>nextLeftLane(currentLane_id);
            leftLane_id = leftLane->lane_id();
            rightLane_id = rightLane->lane_id();
            //If left lane has opposite traffic
            //inform the object about it
            if(rightLane->getInstanceType() == Lane::VEHICLE_LANE)
                  v1->LIU_mode = false;
            else
            {
                  v1->LIU_mode = true;
                  v1->LIU_originalLaneID = currentLane_id;
                  v1->LIU_passingLaneID = leftLane_id;
            }

      }
```

As you can see we first determine which direction we are running. Assuming we have a right side driving system like in most parts of the world, then this will determine which lane is to our left and which is to our right. If there is a logical right lane we should never

attempt to change into the left lane. This is because then it is a better choice to change to the right lane (even if local laws do not allow passing on the right side). However if there are no right lane we should consider the left lane for passing. Later I noticed that I had only designed this function to deal with the simple scenario of a world with only one lane for each direction. A moment I thought I would expand it to cover roads with multiple lanes in the same direction, but since I had no world to test it in I left it as it is. I would not have known if it worked or not in the end without a testing environment.

computeTargetChangingLane()

In this function I added some extra checks to avoid different problems. The first addition I did was to make sure you only considered agents with the same direction ahead of you to be lead cars. This is to prevent meeting agents from treating an overtaking car as a slow moving car on their road.

Example:
```
if(currentAheadVehicles[i].first->facingDir() == obj->facingDir())
```

Only by passing this test were the agents allowed to be considered ahead vehicles. The same restrictions were not added to *currentBehindVehicles* because we would not care what happens behind us currently. For more advanced cases it would have been interesting but in this simple test it was not needed. Later in the code you encounter another special case I added which treats right and left ahead traffic queries. A road segment is usually described with indexes for lanes. On one end of the center everything is positive, 1,2,3, on the other negative, -1,-2,-3. Since we are dealing with overtaking we are certain to be at 1 or -1. So here follows how my code looks for dealing with a mandatory lane change assuming we are on a certain lane.

Example:
```
if(obj->LIU_mode && choice==mandatoryLC)
{
            if(nearestRouteConnectionLane == 1)
            {
                  //If on lane 1
                  targetLane_id = currentLane_id-2;
                  rightLane_id = targetLane_id;
                  leftLane_id = 0;
                  queryAllLeadersOnRoads(obj,
                        rightLane_id,
                        POSITIVE_DIR,
                        aheadRange*200,
                        searchRange,
                        rightAheadVehicles);
                  cout<<"Moving Right <Mandatory>"<<endl;
            }
```

This part also had me thinking about which design to go for. At first I thought I would do it in a way that it could handle multiple lanes, but since I did not have test environments for it and had skipped it in other parts of the code it made sense to keep it simple. If we are performing an overtaking and we receive *mandatoryLC*, we should be on the left lane

during an overtaking. It then checks if you are overtaking on lane 1, and then to reach lane -1 you have to subtract 2 from the current lane. We also query the oncoming traffic but do not treat it until later. The reason for doing it later is just because the default lane changing did it that way. The result of the query is placed in the last argument variable which in this case is *rightAheadVehicles*, and they are placed in pairs of identity number and distance.

A bit later in the code I adjusted a line slightly to make it work both in general lane changing and in overtaking.

Example:
```
(obj->LIU_mode || leftLane->flowDirection()==rpe->road_ptr-
>findLane(currentLane_id)->flowDirection())
```

This means the flow of the traffic has to be the same in both lanes unless we are performing an overtaking. Later in the queries of the oncoming traffic I adjusted the range value to be 1500 meters. I thought this was a reasonable distance to assume that a human driver could see. However we do not consider meeting traffic to be an issue unless they are closer than 500 meters.

Example:
```
if( v1->facingDir() != obj->facingDir())
                    {
                          if(leftAheadVehicles[i].second < 500)
                          {
                                cout<<"Less than 500 meters to
collision"<<endl;
                                leftLane_id=0;
                          }
                          //If we found a meeting car, end check
                          cout<<"Distance to meeting car:
"<<leftAheadVehicles[i].second<<endl;
                          i = static_cast<int>(leftAheadVehicles.size());
                    }
```

As you can see here I weed out the approaching traffic by checking which in my left lane has the opposite direction to the agent who wants to overtake. And if this distance (stored as .second) is less than 500 meters we set *leftLane_id* to zero, which translates into there being no acceptable lane to change to. We also print a message for debugging and stop the query of oncoming traffic because we have received the information we were looking for. The variables may be a bit hard to tell apart in this case. *Obj* is the overtaking agent and *v1* is in this case the closest meeting agent. This might be a bit confusing since in *CheckPassMode()* our *v1* actually was the agent doing the check. Since *CheckPassMode()* was written by me and this part was changed original code I guess *v1* should always be another agent (not the current one).

You might have noticed that *computeTargetChangingLane()* is a rather large function that determines if our desired lane is acceptable or not. The next change I did to the function was to only check if our leader's speed was slower than our own by about 5 km/h. If it was not slower then the agent would not desire to overtake because the speed

is just about right. A normal lane change also looked at the cars behind in the desired lane to change to but I thought this was not particularly needed now. It could be useful to insert a version of this to control that you do not try and overtake if the car behind you is already overtaking, but because of how hard it is to create traffic with varying speeds in Hank I considered it to not happen in my test environment. In the end this function returns which lane we should change to in relative coordinates. If it returns zero it means we should stay on our own lane.

gapAvailableLaneChanging()

I did not change anything in this function actually. While looking through the code it feels like I should have needed to edit it to find gaps in different ways than a normal lane change find gaps. But since it is working I will leave it as it is.

Veh_BehavHCSM.C

Let us start with describing the state machine of the original lane changer. In a function called *preActivity()* it checks if lane changing is turned on. If it is on it runs the considerLaneChanging() function from Path and then depending on if it receives mandatory or no lane change it either sets the state machine to *LCWAITFORGAP* or *LCSTART*. After this it switches over into a different mode where it as long as it has not reached the *LCMOVEOVER* state keeps running *considerLaneChanging()* looking for mandatory or no lane change again. If mandatory again turns up it changes state to *LCWAITFORGAP* again, and if no lane change it returns to the first mode. It is a bit hard to explain but it pretty much just makes sure that if mandatory or no lane change appears we react to it, while if we have the other two modes of limited or discretionary it does not react. In this little cycle of *preActivity()* I added a piece of code myself.

```
        if(_v->LIU_overtake == Vehicle::LANECHANGE && _v->lcState ==
Vehicle::LCMOVEOVER)
        {
              cout<<"Passed"<<endl;
                  _v->LIU_overtake = Vehicle::LANERETURN;
        }
        else if(_v->LIU_overtake == Vehicle::LANERETURN && _v->lcState
== Vehicle::LCMOVEOVER)
        {
                  cout<<"Back at starting lane"<<endl;
                  _v->LIU_overtake = Vehicle::FINISH;
        }
```

What this part does is that it looks at my own state machine (which I will get more into later) and if we have a certain state called *LANECHANGE* it means we actually have changed lane to the oncoming traffic, and because of this we change state to *LANERETURN*. However if we would have the state *LANERETURN* when we reach this it would mean we were back at our original lane and have finished the overtaking and we switch over to *FINISH* state. There is also a default *NOPASS* state that stands for not performing an overtaking and is the default state of the state machine.

A problem with this is that it does not seem to trigger instantly but take a while which means that cars take unnecessary long time to pass. This is mostly visible if you are static in their path and less obvious if you are moving while they pass.

Later a function called *activity()* implements the actual state machine for overtaking is placed. Normally in this function there is a simple function call to *laneChangingStateMachine().*

laneChangingStateMachine()

This is the original lane changing state machine that I tried to expand on to make my own overtaking agents. Depending on a vehicle enum variable it goes into different modes. The enum variables, who are the different states, are called *LCSTART*, *LCWAITFORGAP* and *LCMOVEOVER*.

During *LCSTART* the state machine calls a function *computeAdjacentLanes()* in Path. It stores which *lane ID* is to the agents left and right as *leftlane* and *rightlane*. After that it calls the function *computeTargetChangingLane()* which represent which lane was best. If we found one, we move on to *LCWAITFORGAP* else we keep looking at the lanes.

During *LCWAITFORGAP* we again run *computeAdjacentLanes()* and *computeTargetChangingLane()*. Following this it then runs *gapAvailableLaneChanging()* to determine if there is a gap available. If this returns true then depending on if there either is no leader or if the leader is closer than we would want them to be, we switch over to *LCMOVEOVER*. Several values are at this moment set to make the lane change smooth but I will not go into details about them.

LCMOVEOVER is more or less a PD controller that adjusts the path to move depending on distance left to move. With an offset check it determines when the state is over.

```
if( fabs(_v->path->lookAheadOffset-_v->path-
>laneChangingTargetPathOffset)<0.09)
```

Now that we have an idea about how the original lane change state machine works we are going to look into the overtaking state machine. The lane changing state machine was not changed at all but kept as it was from the beginning. However as I have mentioned before many of the functions it calls had to be changed in different ways.

The first thing that I do before my state machine is run is to check to see if we should run overtaking or normal lane change.

```
	if(_v->LIU_overtake == Vehicle::NOPASS)
	{
		_v->path->checkPassMode(_v);
		if(_v->LIU_mode)
		{
			_v->LIU_overtake = Vehicle::LANECHANGE;
		}
	}
```

The default state in my state machine is *NOPASS*, which means that we are not planning on doing a passing maneuver or overtaking. While in this state we call the function *checkPassMode()* to see if we need to change into overtaking mode depending on what the road looks like. If we want to overtake because there are no lanes in our own direction available to change to, then we change the *LIU_overtake* state into *LANECHANGE*.

```
	//LIU overtaking state machine
	switch(_v->LIU_overtake)
	{
		case Vehicle::NOPASS:
			//If normal lane changes allowed, change lane
			if(_v->laneChangingBehavior==1)
				laneChangingStateMachine(); //using normal
functions
			_v->LIU_mode = false;
			break;
		//If overtaking is to be considered, run these steps
		case Vehicle::LANECHANGE:
			//Change Lane
			//cout<<"Step 1"<<endl;
			_v->LIU_limit = 0;
			_v->desiredSpeed = (_v->fixedSpeed + 5) * MPH2MPS;
			laneChangingStateMachine(); //using edited functions
			break;
		case Vehicle::LANERETURN:
			//cout<<"Step 3"<<endl;
			//Return to starting lane (works both ways alone)
			laneChangingStateMachine(); //using edited functions
			break;
		case Vehicle::FINISH:
			//cout<<"Step 4"<<endl;
			//Restore speed value
			_v->desiredSpeed = _v->fixedSpeed * MPH2MPS;
			//Clear all values
			_v->lcState = Vehicle::LCSTART;
			_v->LIU_mode = false;
			_v->LIU_overtake = Vehicle::NOPASS;
			break;
		default:
			_v->LIU_mode = false;
			_v->LIU_overtake = Vehicle::NOPASS;
			break;
	}
```

As you can see during *NOPASS* we simply run the normal state machine for lane changing if it is allowed (*laneChangingBehavior = 1*). When *LANECHANGE* is the current state we increase the desired speed to pass faster and adjust a limiting value so that we avoid errors. It simply limits certain of the help functions from running parts more than once each pass through the state machine. *LANERETURN* does not do much in the actual state machine but it effects behavior in the help functions in *laneChangingStateMachine().* The *FINISH* state clears up all used variables and sets them to their initial values to be ready to be run again.

It is a bit odd that in ST Software I wanted to adjust acceleration but had to change speed, and in Hank there is no such restriction but I still ended up with using the method of changing speed since I was familiar with it. Looking back at what percievableobject.h offers there seem to be a *double desired_accel* variable.

9.2.4 The end result

After having implemented the agent and tested its performance a few things remained to be dealt with. One thing was the clipping. Currently agents only keep track of the agent closest ahead of them. This means that while agent A can see a stationary car in the way and adjust to change lane, agent B behind them will not see this stationary car until too late when A has started to change lane. The result is that agents can end up clipping through each other's cars. Another thing was how long it took agents to actually change lane back. I inform the agents that it is mandatory to change lane back as soon as they have changed lane once to the meeting traffic's lane, but they still act like agents and take their time. With act like agents I mean that they do not instantly follow orders but consider them to be suggestions on what to do. Mandatory would mean that there is a very good idea to change lane but it does not tell them to change lane instantly but allow for some extra checks first.

There were a few other things that I would have preferred to have solved but I did not manage to. For example the lack of a world editor made me restricted to a rather simple world consisting of nothing else than a long segment of road. If you had a several-road-segment world with intersections the overtaking agent stops working. This is not a huge problem because we do not want people to overtake through intersections.

In the end the overtaking agent was able to detect oncoming traffic and if there was less than 500 meters to them (a number I got from testing and not from any special rules) they avoided to change lane and overtake. It seemed to work well together with their speed of 25 mph (40 km/h, set on creation), but would most likely need adjustments if it were to be able to handle other speeds. In theory we could calculate an expected needed time through the values 40 km/h and 500 meters. I am worried that it might not be linear in how the agents handle the mandatory command while in the meeting traffic lane so it would probably require further testing.

It is worth observing that 500 meters was what worked with the test environment, but that from Table 1 it would seem like 350 meters would have been enough. However the road was set to max speed 35 mph (56 km/h) which could have extended it to closer to 400 meters. It still felt like a real overtaking would be using a shorter distance to overtake than the Hank agents ended up needing.

10 Future Work

There is much work left to do in order to reach the desired level of function of agents in a traffic simulation environment. Among other things, I did not manage to try many of the more advanced equations. I did use the simplest of them but did not find the need or time to implement the more advanced equations and see if they were better or not. Because of this it felt like the first chapter lost a bit of its purpose. I guess you could say that functionality went ahead of finesse. In the future maybe one could try the more advanced equations and see if they give any noticeable improvements.

10.1 Overtaking Agent

There are a few additional things that should be implemented to make the agent more useful in ST Software. The most important one would probably be a lane check that checks for the direction of the lane to the left. The current version assumes there is a single lane in each direction and would not work on multiple lane roads. The second most important addition would be to make the agent handle obvious traffic obstacles like road markings and signs which also remains to be implemented. Nothing prevented me from adding this in ST Software, but the road I was testing I had received from ST Software during one of our many e-mail discussions and I did not have the actual files to change it. So without any road markings I left it for the future. In Hank I never got that far as to deal with road markings until it became too late.

Simulator specific improvements would be in ST Software's case to improve the forward checking during overtaking. As I have mentioned I was forced to turn it off to avoid basic safety routines to kick in during overtaking and force the agent back into its old lane.
In Hanks case the most interesting improvement to do right now would probably be to adjust the vehicle model so it feels more realistic. Currently it is a very simplified version that allows different speeds, but does not feel like real car behavior.

Additional functions that could be useful to cause devastating scenarios could be for example the possibility that an agent decides to overtake on the right lane instead of left. It could also be very interesting to try and design a system that emulates sight for cars. A real driver would not overtake if there were trees in the way of their view of the road structure while the current agents would know the free distance and cheat.

10.2 Traffic Light Agent

I consider the traffic light agent system to be functioning as expected. It does have a few problems that I am unsure if it would be possible to adjust for. One of these problems is when you have two drivers coming to the same traffic light. If you recall the design of the traffic light agent was to reconfigure the time of a cycle of colors and each color's duration when the human driver is at a certain distance to the intersection. To make it certain we knew what color was displayed and for how much longer, a color was set at the same time. So naturally you are forced to adjust to the first driver getting within

range, because adjusting to the other driver would make the closest driver see the light change in peculiar ways.

The current design is using a 150 meters activation range, which means that anything affecting your driving speed after that point is interfering with the result. So it could be interesting to see if it is possible to extend the agent to handle slow moving vehicles between the driver and the traffic light. I would suggest this check is done at perhaps 155 meters and if there are slower agents in front of the driver, their speed will be used to measure time of arrival. In a similar way intersections between the driver and the traffic light during those 150 meters can interfere as well. These would be very hard to adapt to since unless it is also adjusted by a traffic light the time the agent will have to stop at the intersection is unknown. The traffic light had an acceptable function when the human driver kept a steady speed of 50 km/h or 70 km/h, but driving faster could be another issue. While I do not think that too many roads go at 90 km/h or higher and include a traffic light, it could still be possible to expand the agent to handle these cases as well. The trick would be to observe the higher speed, and instead trigger the traffic light change at a longer distance than 150 meters. Another case would be going in 30 km/h, which won't cause any problems with the color but will make each color phase unexpectedly long.

It might be a possible to handle this traffic light agent differently. You could maybe continuously adapt the durations of the cycle to the driver's speed. However I never got around checking if changing cycle time resets the traffic light to red or resets to zero seconds passed on current active color or maybe even if it just keeps counting from where it was. What I mean with the last part is that you could have had 5 seconds duration of green and 3 seconds had already passed. Then you adapt the cycle to only 4 seconds duration but that it kept going from the 3 passed seconds and would only stay green for 1 second longer after change.

Of course it would also be desirable to implement this in Hank to get Hank compared at more areas than just lane changes.

10.3 Meeting Agent

The meeting agent could be improved in similar ways as the overtaking agent. That is to say that it could be improved to deal with signs and road markings and deal with obstacles in the path. A traffic light intersection for example would give a really bad effect on a meeting. But it might also be difficult to know how to adjust for these things in the code.

Another interesting thing to expand on could be how many cars that should be meeting. I think it could be possible to activate two agents if you want to have a meeting between three cars (assuming one of the vehicles is driven by a human driver), but I have not tried this. Finally, it would be a good idea to convert my basic code into a more flexible one where you are able to tell from which direction you want a meeting car to come from.

And as with the traffic lights it would be interesting to get this agent to run in Hank and compare them both.

Just at the end of my thesis work I was asked if it would not have been more interesting to make an agent type that finds their way through a world to a certain point. It sounded interesting to implement but I could not think of a time when you would have any use from it except for making a competition between two agents or an agent and a human to reach a certain place first. Normally you do not want competitions in traffic so I am still unsure of the benefits of implementing one. It is possible that you could replace a human driver with one of these agents to run scenarios unmanned and record agent behavior.

10.4 Hank

Because Hank has a lot less functions implemented yet there are many things that would be interesting to improve. We would want a program that could create worlds as easily as STRoadDesign, but that might be very far into the future. A bit more realistic would be to expect to be able to design a more accurate control system with believable acceleration and breaking and gear shifts. Adding a secondary viewport as a mirror would also be helpful, and maybe even expand the simulator view to 3 monitors, which could cover a 180 degree field of view. A viewport is a camera display that in general does not cover the full screen. This is why it would also be fitting as a mirror.

One could also look into if it is possible to have several humans in the same test environment interacting or if it is restricted to a single human driver. I have no experience at network programming so I can not say if this would be something that could be added or not.

The most important future work for Hank would be to overcome a few important problems. One problem I could not solve was changing the starting lane. If I told my driver to start on the left lane, it started faced against traffic direction and not along it. This meant that I could not test the meeting traffics abilities to overtake, but since I used the same code for both, just adjusting to their different lanes, it should work. But it was not possible to confirm that it works for both lanes. The other enormous problem was that it seemed like the human driver's camera and their logical representation were not bound. This meant that the logical information that agents use was that the human driver was standing still at the starting position. They could never see you move, so they could not interact with an active human driver. My problem was then that Hank had a lot of files and I didn't know which way they had designed it. If it were just a camera position available and a way to position the logical representation it would be fixed fast, but it seemed like nobody knew anything about it and it would have been like finding a needle in a haystack to solve it on my own.

The function that should be worked on first to expand the overtaking agent in Hank would be my *CheckPassMode()* function to check for the actual lane flows and allow for more than one lane in each direction. Currently it just assumes the lane the agent is not driving on is a lane to do an overtaking on, unless the agent has lanes on both sides of their own lane.

11 References

Shladover S. E., & Tan S.-K., Analysis of vehicle positioning accuracy requirements for communication-based cooperative collision warning, *Journal of Intelligent Transportation Systems, vol. 10*, 3, page 131–140, 2006

Cheng B., & Fujioka T. A hierarchical driver model, *Proceedings of IEEE Conference on Intelligent Transportation Systems, ITSC,* page 960-965, 1997

Shamir T., How should an autonomous vehicle overtake a slower moving vehicle: Design and analysis of an optimal trajectory, *IEEE transactions on automatic control, vol. 49,* 4, 2004

Jenkins J. M., & Rilett L. R., Modeling the Interaction Between Passenger Cars and Trucks, *National Technical Information Service,* 2006

Abe G., & Richardson J., Alarm timing, trust and driver expectation for forward collision warning systems, *Applied Ergonomics, vol. 37,* 577-586, 2006

Willemsen P. J., *Behavior and scenario modeling for real-time virtual environments,* PhD thesis, University of Iowa, 2000

McLoughlin H. B., Michon J. A., van Winsum W., & Webster E., GIDS intelligence, in J. A. Michon, ed. *Generic Intelligent Driver Support,* chapter 6, page 89-111, Bristol, PA: Taylor & Francis, 1993

Liu B.-S., Association of intersection approach speed driver characteristics, vehicle type and traffic conditions comparing urban and suburban areas, *Accident Analysis and Prevention, vol. 39,* page 216-223, 2007

Broughton K. L. M., Switzer F., & Scott D., Car following decisions under three visibility conditions and two speeds tested with a driving simulator, *Accident Analysis and Prevention, vol. 39,* page 106-116, 2007

Wills A. R., Watson B., & Biggs H. C., Comparing Safety climate and factors as predictors of work-related driving behavior, *Journal of safety research, 37,* page 375-383, 2006

Bar-Gera H., & Shinar D., The tendency of drivers to pass other vehicles, *Transport research, part F 8,* page 429-439, 2005

Vanderhaegen F., Chalmé S., Anceaux F., & Millot P., Principles of cooperation and competition: application to car driver behavior analysis, *Cognition, Technology and Work, vol. 8,* 3, page 183-192, 2006

Hoban C. J., & McLean J. R., Progress with Rural Traffic Simulation, *Proceedings of the Australian Road Research Board, vol. 11,* Part 4, page 23-58, 1982

ST Software, User's manual, 2005

US Department of Transportation – Federal Highway Administration, *Surrogate Safety Measures From Traffic Simulation Models, chapter 4,* http://www.tfhrc.gov/safety/pubs/03050/, Accessed 2007-07-09

Virtual Terrain Project, *Vehicles and traffic,* http://www.vterrain.org/Culture/vehicles.html, Accessed 2007-07-09

ST Software, *Car driving simulation software,* http://www.stsoftware.nl/StRoadDesign.html, Accessed 2007-07-09

VIRES Simulationstechnologie, *OpenDRIVE,* http://www.opendrive.org/, Accessed 2007-07-09

Swedish Road Administration, *Vägutformning 94, Version S-2, Del 13 Trafiksignaler,* Publ. 2002:123, http://www.vv.se/templates/page3____11426.aspx, Accessed 2007-07-09

LINKÖPING UNIVERSITY
ELECTRONIC PRESS

På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Linus Gustavsson